Community Search with K-Core and K-Truss

Xiang Xu
The Chinese University of Hong Kong
1155107785@link.cuhk.edu.hk

Yu Chen
The Chinese University of Hong Kong
1155107766@link.cuhk.edu.hk

ABSTRACT

Based on bountiful data from online social networks, community search in response to users' queries becomes a recently emerging topic. Given a query vertex in a graph, the problem is how to find meaningful communities that the vertex belongs to in an efficient manner. In this paper, we study *community search* problem by implementing k-core and k-truss methods. Compact index structures for both methods are introduced to support more efficient community search. To demonstrate their application on real data, extensive experiments on Facebook Dataset are conducted and comparisons between their algorithm performance are made as well.

1 INTRODUCTION

In real-life networks, including the Internet, biological networks and social networks, vertices in the graph representation tend to form *community structures*. *Community structures* represent groups of nodes whose connections are dense to each other within a large network. For some analytical tasks, *community detection*[2, 12–14, 18] is conducted to discover community structures with interpretable meanings: in a Facebook social network, a community structure is likely to be a real-life social community. The importance of *community detection* work led to the emergence of *community search* problem. *Community search* problem returns community results in regard with the user's query vertex [16]. We focus on *community search* problem in our project due to the following reasons:

(1) Instead of global search, local search in a graph can be conducted if a query input is given, community search can largely reduce the cost in community analysis in community detection problem [6].
(2) Community search gives more personalized results, which is more consistent with real-life applications [16].

The definitions of the community vary according to different data structure used to solve community search problem. Among these different definitions, the problem could all be simply defined as follows [6, 9, 16]: *Given a query vertex in a graph, the goal is to find meaningful communities that the vertex belongs to.*

Previous works propose multiple methods to tackle the problem. Both traditional algorithms and graph neural networks are discussed to solve the community search problem. Traditional algorithms adopt data structures and build new indices including k-clique [7], quasi-clique [1], k-core [3, 4, 15], k-truss structures [5, 9, 17], etc.

This term we mainly concentrated on k-core and k-truss models. Experiments are done to better understand the algorithms and comparisons are made to draw conclusion.

The organization of this paper is as follows. We provide problem formulation in *Section 2*. In *Section 3* and *Section 4*, we introduce index construction and query algorithms for k-core and k-truss respectively. Experiment results are reported in *Section 5* and we discuss and conclude our work in *Section 6*.

2 PROBLEM DEFINITION

We consider an undirected, unweighted simple graph $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges. Given such a graph $G = (V, E)$, a query vertex $v_q \in V$ and an integer k, find all potential communities containing v_q based on k-core model and k-truss model. Table 1 summarizes the notations used in this paper.

$G = (V, E)$	A graph with vertex set V and edge set E.
$G[H]$	The subgraph of G induced by a set of vertex H.
$deg_G(v)$	The degree of vertex v in G.
$N(v)$	The set of neighbors of a vertex v.
d_{max}	The maximum vertex degree in G.

Table 1: Notations

2.1 Preliminaries for K-Core

Seidman [15] introduced the concept *core* in 1983: with $n = |V|$ vertices and $m = |E|$ edges in a graph $G = (V, E)$, $G[H]$ is a subgraph of G containing node set H and the edges induced by H.

Definition 2.1 (K-Core). The set W is called *k-core* or *a core of order k* iff $\forall v \in H : deg_{G[H]}(v) >= k$ and H is the maximum subgraph satisfying this property.

Definition 2.2 (Core). The *core number* or *coreness* of vertex v is the highest order of a core that contains v.

2.2 Preliminaries for K-Truss

We then introduce several definitions to formulate k-truss concept. *k-truss community* is defined on the basis of the definitions of support and triangle connectivity [9]. For the completeness of the report, all notations and definitions are from [9] to demonstrate the process in formulating *k-truss community*.

Definition 2.3 (Triangle). A triangle in G is a cycle of length 3. Let $u, v, w \in V$ be the three vertices on the cycle, and we denote this triangle by $\triangle_{uvw}$.

Definition 2.4 (Support). The support of an edge $e(u, v) \in E$ in G, denoted by $sup(e, G)$, is defined as $|\{\triangle_{uvw} : w \in V\}|$. When the context is obvious, we replace $sup(e, G)$ by $sup(e)$.

Definition 2.5 (Triangle Adjacency). Given two triangles $\triangle_1, \triangle_2$ in G, they are adjacent if $\triangle_1$ and $\triangle_2$ share a common edge, which is denoted by $\triangle_1 \cap \triangle_2 \neq \emptyset$.

Definition 2.6 (Triangle Connectivity). Given two triangles $\triangle_s, \triangle_t$ in G, $\triangle_s$ and $\triangle_t$ are triangle connected, if there exist a series of triangles $\triangle_1, \ldots, \triangle_n$ in G, where $n \geq 2$, such that $\triangle_1 = \triangle_s$, $\triangle_n = \triangle_t$ and for $1 \leq i < n$, $\triangle_i \cap \triangle_{i+1} \neq \emptyset$.

Definition 2.7 (K-Truss Community). Given a graph G and an integer $k \geq 2$, G' is a k-truss community, if G' satisfies the following three conditions:

(1) K-Truss. G' is a subgraph of G, denoted as $G' \subseteq G$, such that $\forall e \in E(G')$, $sup(e, G') \geq (k-2)$;

(2) Edge Connectivity. $\forall e_1, e_2 \in E(G')$, $\exists \triangle_1, \triangle_2$ in G' such that $e_1 \in \triangle_1$, $e_2 \in \triangle_2$, then either $\triangle_1 = \triangle_2$, or $\triangle_1$ is triangle connected with $\triangle_2$ in G';

(3) Maximal Subgraph. G' is a maximal subgraph satisfying conditions (1) and (2). That is, $\nexists G'' \subseteq G$, such that $G' \subset G''$, and G'' satisfies conditions (1) and (2).

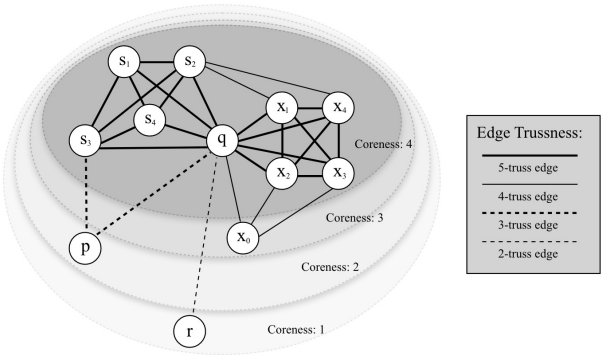


Figure 1: An illustrating example of k-core and k-truss on the same graph

Definition 2.8 (Subgraph Trussness). The trussness of a subgraph $H \subseteq G$ is the minimum support of an edge in H, denoted by $\tau(H) = \min\{sup(e, H) : e \in E(H)\}$.

Definition 2.9 (Edge Trussness). The trussness of an edge $e \in E(G)$ is defined as $\tau(e) = \max_{H \subseteq G}\{\tau(H) : e \in E(H)\}$.

Example: To illustrate the definitions more clearly, we take the graph in Figure 1 as a concrete example. Focus on two subgraphs $G[\{q, s_1, s_2, s_3, s_4\}]$ and $G[\{q, x_1, x_2, x_3, x_4\}]$. As we observe, they are two fully connected communities and each edge has 5 trussness. Thus, we obtain two 5-truss communities both containing query vertex q in Figure 1. Using $C = \{q, s_1, s_2, s_3, s_4\}$ as an example, each edge inside C is contained in at least three triangles. In terms of triangle connectivity, any two edges in C are connected through adjacent triangles as we defined before. C is also the maximal subgraph, and cannot include nodes $\{x_1, x_2, x_3, x_4\}$ to merge into one large 5-truss community. Besides, one interesting observation is that the vertex q participates in both truss communities.

3 K-CORE COMMUNITY

In this section, we firstly introduce a basic core decomposition method proposed by *Batagelj* and *Mrvar* [3]. Then a bin-sort method, a faster development based on the basic algorithm introduced in 2003 is discussed [4]. Finally we introduce how to obtain results of a community given a query node input based on the work of *Cui et al.* [6].

3.1 Basic Core Decomposition

Basic core decomposition is based on this property: if from a given graph $G = (V, E)$, all vertices whose degree is less than k, along with the edges induced by them are deleted, the remaining subgraph is the k-core [3].

As shown in Algorithm 1, basic k-core decomposition keeps an array recording core number for each vertex. In line 1-2, the degree of each node is obtained and sorted in an ascending order. Initially, the core number of each node is set to be its degree (line 3-4). In line 5-7, it iterates through each node following the degree ordering and compares the selected node's degree with its neighbors' degrees. When the neighbor's degree is larger, this neighbor's core number decrements 1. The table keeping core numbers is reordered at the end of every iteration (line 8).

The intuition behind this algorithm is that for each node, if its neighbor's degree is larger than the selected node's degree, it means its neighbor cannot form a connected subgraph whose degrees are all larger than or equal to its neighbor's own degree. Therefore, the maximum of the possible core number of this neighbor is n-1, suppose the original degree is n. The subtracted 1 represents the degree contributed from the selected node.

Algorithm 1: Basic Core Decomposition

Input: $G = (V, E)$
Output: $core(v)$ for each $v \in V$

```
1 compute deg(v) for each node v ∈ V;
2 sort all the nodes in ascending order of their degree;
3 foreach v ∈ V in the order do
4     core(v) ← deg(v);
5     foreach u ∈ N(v) do
6         if deg(u) > deg(v) then
7             deg(u) ← deg(u) − 1;
8             reorder V accordingly;
```

Example: Suppose we want to decompose the graph in Figure 1. Algorithm 1 first computes the degree for each node and sorts them in ascending order. The result is $\{r, p, x_0, s_1, s_4, s_3, x_1, x_2, x_3, x_4, s_2, q\}$. Since r's only neighbor p has a larger degree than r, r's core number is updated as 1. No further update of r's core number is needed. Hence, the coreness of r is 1. Meanwhile, degree of p decrements, indicating that r is deleted from the graph. For p's deletion, s_3's degree is set to 4 and the coreness of p is set to 2. x_0's neighbors q, x_2, x_3 all have a larger degree than x_0. Hence, the three edges connecting to x_0 are deleted and degree of q, x_2, x_3 are subtracted correspondingly. After the remaining iterations, we may end up with the remaining graph $\{s_1, s_4, s_3, x_1, x_2, x_3, x_4, s_2, q\}$ a 4-core.

3.2 Bin-sort Based Core Decomposition

Bin-sort based k-core decomposition shares the same idea with basic k-core decomposition with a speed improvement in reordering the updated core numbers of each node [4]. It updates the core number and finishes reordering more efficiently with 4 arrays: *deg, bin, vert, pos.* The algorithm can be divided into 2 steps: *initialization phase* (line 1-18) and *core decomposition phase* (line 19-28) [4].

Algorithm 2: Bin-sort Based K-core Decomposition

Input: $G = (V, E)$
Output: $core(v)$ for each $v \in V$

1 compute $deg(v)$ for each node $v \in V$;
2 $md \leftarrow maximum\ degree$;
3 **foreach** d *from* 0 *to* md **do**
4 $bin(d) = 0$;
5 **foreach** $v \in V$ **do**
6 $bin(deg(v)) \leftarrow bin(deg(v)) + 1$;
7 start $\leftarrow 1$;
8 **foreach** $from$ 0 *to* md **do**
9 $num \leftarrow bin(d)$;
10 $bin(d) \leftarrow start$;
11 $start \leftarrow start + num$;
12 **foreach** $v \in V$ **do**
13 $pos(v) \leftarrow bin(deg(v))$;
14 $vert(pos(v)) \leftarrow v$;
15 $bin(deg(v)) \leftarrow bin(deg(v)) + 1$;
16 **foreach** d *from* md *to* 1 **do**
17 $bin(d) \leftarrow bin(d-1)$
18 $bin(0) \leftarrow 1$;
19 **foreach** i *from* 1 *to* n **do**
20 $v \leftarrow vert(i)$;
21 **foreach** u *in* $N(v)$ **do**
22 **if** $deg(u) > deg(v)$ **then**
23 $du \leftarrow deg(u)$; $pu \leftarrow pos(u)$;
24 $pw \leftarrow bin(du)$; $w \leftarrow vert(pw)$;
25 **if** $u \neq w$ **then**
26 $pos(u) \leftarrow pw$; $vert(pu) \leftarrow w$;
27 $pos(w) \leftarrow pu$; $vert(pw) \leftarrow u$;
28 $bin(du) \leftarrow bin(du) + 1$; $deg(u) \leftarrow deg(u) - 1$;

In *initialization phase*, *deg* array firstly keeps the degree for each node without sorting and compute the maximum degree (line 1-2). Each *bin* contains the nodes of the same degree. The index of each *bin* is exactly the degree. Line 3-6 counts the number of vertices belonging to each *bin*, which have the same degree. The starting position of each *bin* is recorded in line 7-11.

According to the recorded starting position, the vertices are stored in *vert* array and *pos* array stores the vertices as array index with its corresponding position (Line 12-15). When we update *vert* and *pos* array, *bin* array is increased and line 16-17 resets the values for *bin*.

In *core decomposition* part, the algorithm still aims to delete all vertices of degree of smaller degree and assign the core number to the remaining nodes (line 19-28). The difference between basic decomposition and this method is that the sorting is integrated to the recursive deletion: when a vertex's degree is decreased, it is moved to the left *bin* to the original *bin*, which should contain nodes of smaller degree.

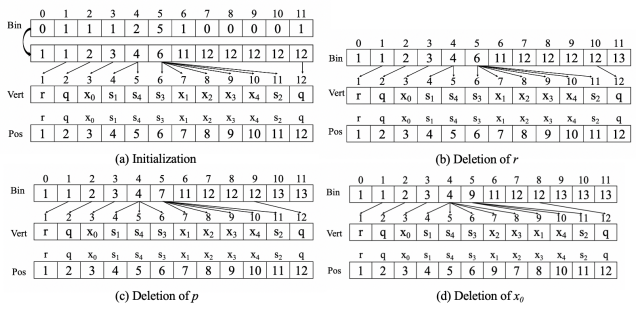


Figure 2: An illustrating example of bin-sort based k-core decomposition

Compared with traditional reordering methods' time complexity of $O(nlogn)$, bin-sort based k-core decomposition's time complexity is linear because the core number updating is equivalent to visiting each edge at most twice [4].

Example: Suppose we decompose the graph in Figure 1 with Bin-sort based algorithm. Algorithm 2 first computes the degree for each node and put down the maximum degree as 11. As shown in Figure 2(a), a *bin* array with 12 elements is initially created with each bucket containing the number of nodes of the degree number. *Bin* is then updated to be the position index of array *vert*. With the position index in *bin*, *vert* and *pos* arrays are constructed. In the sorting phase, r's neighbor q has larger degree, $bin(11)$ is set to 13 and degree of q is set to 10 (shown in Figure 2(b)). As is shown in Figure 2(c), because p's neighbors s_3, q both have larger degree, $bin(5)$ and $bin(10)$ increases to 7 and 13 respectively (each increments 1). Degrees of s_3 and q both subtract 1 to be 4 and 9 respectively. Deletion of node x_0 firstly adjust the records of q: degree of q is updated to 8. The algorithm then swaps x_2,x_1 and x_3, x_1 in *vert* and *pos*. The detailed result is shown in Figure 2(d).

3.3 Query Search with K-Core Decomposition Index

Algorithm 3: Query Search with K-core Decomposition Index

Input: $G = (V, E)$, v_0,k
Output: $G[H]$

1 $queue.enque(v_0)$; $C \leftarrow \emptyset$;
2 **while** *queue is not empty* **do**
3 $v \leftarrow queue.deque()$;
4 $C \leftarrow C \bigcup \{v\}$;
5 **foreach** $(v, w) \in E$ **do**
6 **if** w *is not visited and* $core_G(w) >= k$ **then**
7 $queue.enque(w)$;
8 **return** C;

With k-core decomposition results, query search problem is simplified to obtaining nodes whose core number is larger than k and is in the same community with the query node. Therefore, performing a *BFS* traversal from the query node and including

the nodes with core number larger than k can generate the result community as is shown in Algorithm 3 [6]. Similar to *BFS*, the time complexity of Algorithm 3 is $O(m+n)$ where m is the number of edges and n is the number of nodes in the graph.

Example: Suppose we query the community with query node q and $k = 4$ from graph 1. Initially, *queue* is $\{q\}$. Then v is q and the result set C is $\{q\}$. In all the neighbors of q,$\{r, p, x_0\}$ do not have a node with core number larger than or equal to 4. Nodes $\{s_1, s_2, s_3, s_4\}$ and $\{x_1, x_2, x_3, x_4\}$ all have core number 4. Thus these nodes are put to *queue*. Then, all the nodes in *queue* are dequeued to check whether they have other unvisited neighbors and are put to the result set C iteratively. Finally, $C = \{s_1, s_2, s_3, s_4, x_1, x_2, x_3, x_4, q\}$ is returned.

4 K-TRUSS COMMUNITY

In this section, we introduce another community model based on the k-truss. The k-truss utilizes predefined edge triangles to model the connectivity among nodes [9]. Given a graph G, the k-truss communities of G are all maximum subgraphs in which every edge is contained in at least $(k-2)$ triangles within the subgraph [5, 9]. We first introduce a simple k-truss index and propose an algorithm for k-truss community search based on the index. Due to the limitations of simple k-truss index, we then introduce a novel Triangle Connectivity Preserved Index [9], or TCP-Index in short, which is designed to avoid the computational problems in Algorithm 5.

4.1 Simple K-Truss Index

We first introduce a simple k-truss index and its corresponding algorithm for k-truss community search.

Algorithm 4: Truss Decomposition

Input: $G = (V, E)$
Output: $\tau(e)$ for each $e \in E$

```
1  k ← 2;
2  compute sup(e) for each edge e ∈ E;
3  sort all the edges in ascending order of their support;
4  while ∃e such that sup(e) ⩽ (k − 2) do
5      let e = (u, v) be the edge with the lowest support;
6      assume, w.l.o.g, deg(u)(v);
7      for each w ∈ N(u) and (v, w) ∈ E do
8          sup((u, w)) ← sup((u, w)) − 1;
9          sup((v, w)) ← sup((v, w)) − 1;
10         reorder (u, w) and (v, w) according to their new
             support;
11     τ(e) ← k, remove e from G;
12 if not all edges in G are removed then
13     k ← k + 1;
14     goto Step 4;
15 return {τ(e)|e ∈ E};
```

K-Truss Index Construction. Truss decomposition helps us obtain edge trussness [17]. As shown in Algorithm 4, we first initialize k as 2. As preparation of the algorithm, the *support* for each edge is needed and we sort all the edges according to their *support* in a ascending order. The essential step of the algorithm is that the algorithm iteratively removes a lowest support edge $e(u, v)$ if $sup(e) \leq k-2$ [9]. At the same time, for those who form a triangle with e, we also decrement their *support* and reorder them after adjustment. Afterwards, the trussness k is assigned to the removed edge e. The terminating condition for this iteration is when all edges e with $sup(e) \leq k-2$ are removed. Therefore, by this method, we finally decompose the graph and obtain the trussness of all edges. Figure 1 shows the final result of truss decomposition by indicating the trussness of each edge.

Algorithm 5: Query Processing Using K-Truss Index

Input: $G = (V, E)$, an integer k, query vertex v_q
Output: k-truss communities containing v_q

```
1  visited ← ∅; l ← 0;
2  for u ∈ N(v_q) do
3      if τ((v_q, u)) ≥ k and (v_q, u) ∉ visited then
4          l ← l + 1; C_l ← ∅; Q ← ∅;
5          Q.push((v_q, u)); visited ← visited ∪ {(v_q, u)};
6          while Q ≠ ∅ do
7              (x, y) ← Q.pop(); C_l ← C_l ∪ {(x, y)};
8              for z ∈ N(x) ∩ N(y) do
9                  if τ((x, z)) ≥ k and τ((y, z)) ≥ k then
10                     if (x, z) ∉ visited then
11                         Q.push((x, z));
                           visited ← visited ∪ {(x, z)};
12                     if (y, z) ∉ visited then
13                         Q.push((y, z));
                           visited ← visited ∪ {(y, z)};
14 return {C_1, ..., C_l};
```

Query Processing. As stated in [9], Algorithm 5 illustrates how to conduct k-truss community query based on the simple index. The input of the algorithm is an integer k and a query node v_q, then it is expected to return us searched k-truss communities containing v_q. To begin with, the algorithm looks into each incident edge (v_q, u) to find those with $\tau((v_q, u)) \geq k$ and push them into a queue Q one by one. The rationale behind is that such edge (v_q, u) can potentially be the seed edge to form a new truss community C_l [9]. Then *BFS* traversal search is performed to expand this community C_l with satisfying conditions. The formation of C_l will not terminate until Q becomes empty. Afterwards, the same process is performed to check next potential seed edge for a new community C_{l+1} until we obtain all remaining k-truss communities.

Example: Here we provide a modified example from [9] to illustrate the querying process. Assume our purpose is to find 5-truss communities with q as querying node in Figure 1. Algorithm 5 first picks an edge with trussness 5 and connected to querying vertex q as seed edge. For instance, we add edge (q, x_1) into Q. The *BFS* expansion then starts with edge (q, x_1), iteratively searching for neighboring edges like (q, x_2) which also satisfy trussness condition. All of searched edges are inserted into C_1 which is generated by seed edge (q, x_1). The termination of *BFS* expansion means the

end of formation of C_1 community. The algorithm then checks other unvisited seed edges to form more communities if there exist. In our example, another community C_2 containing $\{q, s_1, s_2, s_3, s_4\}$ is also expected to be found.

Algorithm 6: TCP-Index Construction

Input: $G = (V, E)$
Output: TCP-Index $\mathcal{T}_x$ for each $x \in V$

Perform truss decomposition for G;
for $x \in V$ **do**
  $G_x \leftarrow \{(y,z)|y,z \in N(x), (y,z) \in E\}$;
  for $(y,z) \in E(G_x)$ **do**
    $w(y,z) \leftarrow \min\{\tau((x,y)), \tau((x,z)), \tau((y,z))\}$;
  $\mathcal{T}_x \leftarrow N(x)$;
  $k_{max} \leftarrow \max\{w(y,z)|(y,z) \in E(G_x)\}$;
  for $k \leftarrow k_{max}$ *to* 2 **do**
    $S_k \leftarrow \{(y,z)|(y,z) \in E(G_x), w(y,z) = k\}$;
    for $(y,z) \in S_k$ **do**
      if *y and z are in different connected components in* $\mathcal{T}_x$ **then**
        add (y,z) with weight $w(y,z)$ in $\mathcal{T}_x$;
return $\{\mathcal{T}_x|x \in V\}$;

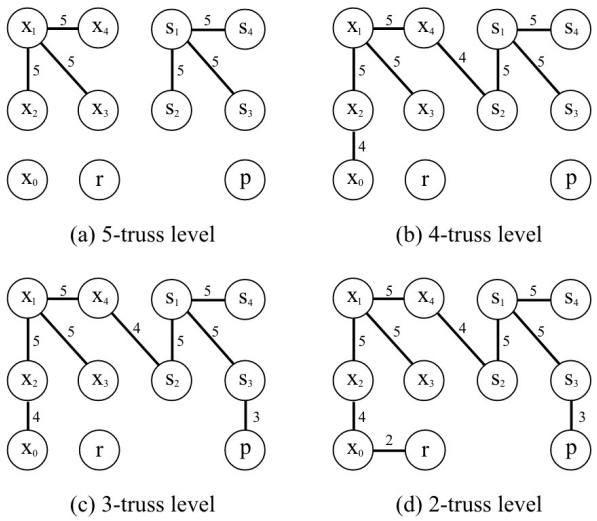


Figure 3: TCP-Index construction of vertex q

4.2 A Novel TCP-Index

TCP-index is a more advanced way to construct the truss index. In order to reduce the unnecessary and excessive computational overhead, the novelly proposed algorithm removes two cases of edges: one is unnecessary access of disqualified edges and the other one is repeated access of qualified edges [9]. Therefore, compared to simple k-truss index, TCP-Index is much more efficient.

TCP Index Construction. As Algorithm 6 outlines, the main idea of the TCP Index Construction was discussed in [9]. We first perform truss decomposition for the given graph G. Then, for each vertex x belonging to V, we construct G_x with $V(G_x) = N(x)$ and $E(G_x) = \{(y,z)|(y,z) \in E(G), y,z \in N(x)\}$ [9]. In addition, the weight of each edge $(y,z) \in E(G_x)$ is the minimum value among the trussnesses of edges (x,y), (x,z) and (y,z). After initializing TCP-Index $\mathcal{T}_x$ for node x as $N(x)$, we assign k_{max} as the largest weight inside G_x. For k iterating from k_{max} to 2, every time we pick S_k, which is a subset of $E(G_k)$. Every edge in S_k has weight k. As shown in Algorithm 6, for every edge (y,z) belonging to S_k, if y and z are in distinctive connected components of $\mathcal{T}_x$, we insert edge (y,z) together with its corresponding weight into $\mathcal{T}_x$. Our final goal is to find the maximum spanning forest by looping all $x \in V$.

Example: In order to illustrate the workflow of the construction of TCP-Index, an example modified from [9] is provided here. Figure 3 presents the step-by-step index building process for node q in Figure 1. Figure 3(a) is the initial graph if we put into edges with trussness 5. As shown in Figure 3(b), adding edge (x_4, s_2) is enough to indicate that two components $\{x_1, x_2, x_3, x_4\}$ and $\{s_1, s_2, s_3, s_4\}$ are reachable through 4-trussness edge according to triangle adjacency. By performing Algorithm 6 repetitively for all $x \in V$, we eventually obtain the compete TCP-Index as shown in Figure 3(d).

Algorithm 7: Query Processing Using TCP-Index

Input: $G = (V, E)$, an integer k, query vertex v_q
Output: k-truss communities containing v_q

$visited \leftarrow \emptyset$; $l \leftarrow 0$;
for $u \in N(v_q)$ **do**
  if $\tau((v_q,u)) \geq k$ ***and*** $(v_q,u) \notin visited$ **then**
    $l \leftarrow l+1$; $C_l \leftarrow \emptyset$; $Q \leftarrow \emptyset$;
    $Q.push((v_q,u))$;
    while $Q \neq \emptyset$ **do**
      $(x,y) \leftarrow Q.pop()$;
      if $(x,y) \notin visited$ **then**
        compute $V_k(x,y)$;
        for $z \in V_k(x,y)$ **do**
          $visited \leftarrow visited \cup \{(x,z)\}$;
          $C_l \leftarrow C_l \cup \{(x,z)\}$;
          if *the reversed edge* $(z,x) \notin visited$ **then**
            $Q.push((z,x))$;
return $\{C_1, \cdots, C_l\}$;
Procedure compute $V_k(x,y)$;
return $\{z|z$ is connected with y in $\mathcal{T}_x$ through edges of weight $\geq k\}$;

Query Processing Using TCP-Index. With holding the TCP-Index, we can now do query processing by performing Algorithm 7. Its main idea is cited from [9] and the mechanism is quite similar to Algorithm 5. Algorithm 7 first picks seed edge and then expanding the corresponding community set in *BFS* manner. The difference lies in how to utilize the TCP-Index for more efficient search. For an unvisited edge (x,y), it searches k-level connected vertex set

$V_k(x, y)$ obtained from $\mathcal{T}_x$ [9]. For each z belonging to $V_k(x, y)$, we add edge (x, z) into community C_l. If the reversed edge (z, x) is unvisited, it is pushed into Q for z-centered community expansion using $\mathcal{T}_z$ [9]. After all seed edges of v_q are visited, the algorithm outputs the final result of multiple k-truss communities if possible.

Example: Suppose we desire to search 5-truss communities with querying node q, the algorithm first visit (q, x_1) in order since its trussness is 5. From TCP-Index $\mathcal{T}_q$, we obtain $V_5(q, x_1) = \{x_1, x_2, x_3, x_4\}$ because they form a connected component with edges weighted as 5. Obviously, these four nodes belong to the same 5-truss community. To further reconstruct the remaining edges of this community, we reverse the edge (q, x_2) to (x_2, q). By querying x_2's TCP-Index again, we know x_2 is also connected with vertex set $\{q, x_1, x_3, x_4\}$. So that in this way, we can recover the full community with less computational cost since each edge in the community is accessed exactly twice [9].

5 EXPERIMENTS

We present our experimental results in this section. *Section 5.1* shows the basic information of our dataset used for experiments. Then in *Section 5.2* and *Section 5.3*, we discuss the evaluation results for k-core and k-truss model respectively. We make comparison in *Section 5.4* regarding to their algorithm performance. Finally, we provide a case study for a deeper understanding of the performance of both algorithms in *Section 5.5*.

5.1 Facebook Dataset

We use Facebook social network from the Stanford Network Analysis Project [11]. Table 2 shows its basic information.

This dataset contains the ground-truth communities of 10 nodes. What needs us to pay attention to is that the ground-truth is the community in ego-neighborhood of one layer of each node. However, the results derived from the core or truss search method are generated from the whole graph. Hence, we obtained the intersection of ego nodes' neighbors and the searched nodes when doing performance evaluation.

Dataset	Vertices	Edges	Average Degree
Facebook	4039	88234	43.691

Table 2: Dataset used in our experiments.

5.2 Evaluation for K-Core

In k-core community evaluation, under one specific querying node q and input parameter k, we adopt the highest $F1$ score to measure the searching performance. It means that a correspondence is assigned between the most fitted community of multiple ground-truth communities and the searched community. That corresponding community is considered as the best matching community for q because it achieves the best F1 score. Since parameter k is not deterministic, in order to find the best $F1$ score, we iterate all possible k values for the querying node.

For consistency with ground-truth dataset, we do intersection with ego-neighborhood of querying node for all involved communities. Figure 4 shows the experiment results for k-core algorithm.

5.3 Evaluation for K-Truss

In k-truss query search, the parameter k can also vary according to the user's input and lead to different search results. However, we do not know which k gives the optimal result. The evaluation method to determine the best k for the query node is used by Huang et al. [9].

We first denote the searched communities as $C = \{C_1, \ldots, C_i\}$, and the ground-truth communities as $\overline{C} = \{\overline{C}_1, \ldots, \overline{C}_j\}$. We adopt F_1 score to evaluate the alignment between a searched community C and a real one $\overline{C}$. Because the correspondence between communities in C and $\overline{C}$ is not known, we calculate the optimal match through linear assignment [11] by maximizing

$$\max_{f: C \to \overline{C}} \frac{1}{|f|} \sum_{C \in dom(f)} F_1(C, f(C)), \tag{1}$$

where f is a (partial) correspondence between C and $\overline{C}$.

For each query node, the result C_k denotes a group of communities returned when querying with threshold k. The ground-truth provides a group of community $\overline{C_k}$. In order to find the matching between C_k and $\overline{C_k}$, for each community in C_k, we compute F_1 score with each community in $\overline{C_k}$. The highest F1 score indicates the best matching. Finally, we obtain the best matching for each community in C_k, with F_1 score denoted as $\mathcal{X}_k = \{X_{k1}, ..., X_{ki}\}$. The average score of $\mathcal{X}_k$ represents the quality of C_k. Therefore, from $C_2, C_3, ..., C_k$, the largest average F_1 score determines the threshold k. Figure 4 shows the experiment results for k-truss algorithm.

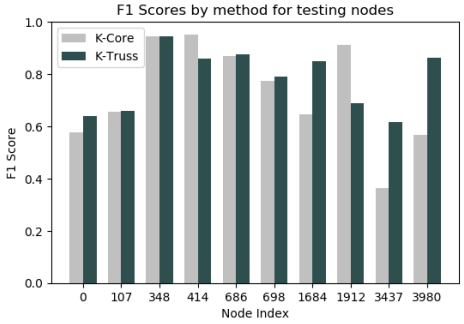


Figure 4: F1 Score

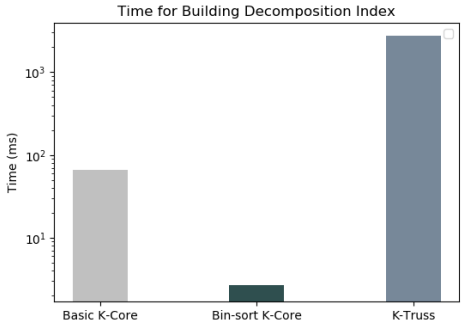


Figure 5: Time for Building Index

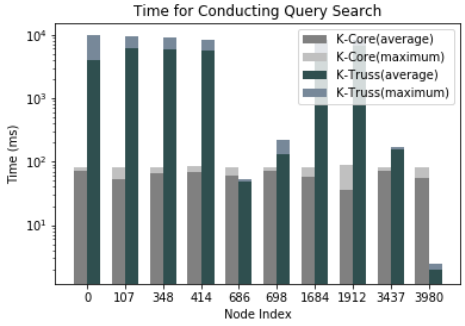


Figure 6: Time for Query Search

5.4 Performance Comparison

From the F1 performances of 10 query nodes, we observe that there is no single algorithm dominating the evaluation for all 10 nodes. For 8 nodes, k-truss performs better than k-core. We could draw a conclusion that according to the 10 nodes, k-truss has a better performance in accuracy under most cases. However, the comparison of which algorithm is better in searching communities is difficult due to the following reasons:

(1) K-truss searches multiple communities while k-core only obtains one community. This leads to different evaluation procedures for two algorithms. Hence, the simple comparison of F1 score does not tell the whole story.
(2) Ground-truth data have multiple communities for an ego-node, which is more consistent with algorithms searching for multiple communities.

Next, we also record the time performance for each algorithm. Regarding the time complexity analysis, as is shown in figure 5, it is obvious in our experiment that core decomposition with bin-sort method is much faster than the simple decomposition method. Figure 6 shows the query time of each algorithm. Normally, k-truss requires much more time to get the result than k-core, but its time performance varies. For example, searching time for query node 3980 and 686 is small due to the community size. The time complexity in our experiments are generally consistent with the results noted in the papers [4, 9].

5.5 Case Study

In order to have a high-level understanding of the searched communities using different algorithms, we plot Figure 7 to visualize the output result when querying with node 698 as case study. Figure 7(a) shows 5-truss communities with querying node 698 on Facebook dataset. Figure 7(b) is the searched 17-core community with the same input vertex. We chose 5 as k for truss and 17 for core because they achieve the best F1 score in the experiment discussed in the previous section. For the purpose of clearer presentation, we only include the nodes within the *ego neighborhood* of node 698.

Visualization of Ego Node In Figure 7(a), green, orange and brown clusters are three communities searched by 5-truss. The grey nodes are those who does not belong to any community, according to the query results. Figure 7 shows one community searched by 17-core, which is a green cluster. Figure 7(c) provides a visualization of the ground truth communities of node 698, with five colors (green, orange, brown, pink and blue) representing five real communities.

We have the following observations from this case:

(1) 5-truss result with node 698 clearly shows that truss query algorithm is able to obtain multiple communities. Some communities (the orange and the brown cluster) are quite consistent with the ground-truth communities.
(2) The largest community (the green cluster in Figure 7(a)) from 5-truss is a superset of some real communities (the blue and green clusters in Figure 7(c)). The reason behind lies in the using of triangle adjacency as measurements of truss community's cohesiveness and connectivity, which leads to the merge of two "connected" components.
(3) The community searched by 17-core is consistent with the largest community in ground-truth communities (the green cluster in Figure 7(c)). It visualizes why k-core achieves fair F1 score in the previous section.
(4) The nodes inside the green cluster of Figure 7(b) have relatively high degree. In this sense, k-core algorithm shows its ability to detect edge-dense community.
(5) K-core algorithm has limitations in applications requiring multiple community detection. In Figure 7(b), we can clearly see 17-core only obtains one community and the other communities will not be searched. Therefore, in terms of the quantity of valid nodes searched, k-truss algorithm outperforms k-core.

6 DISCUSSION & CONCLUSION

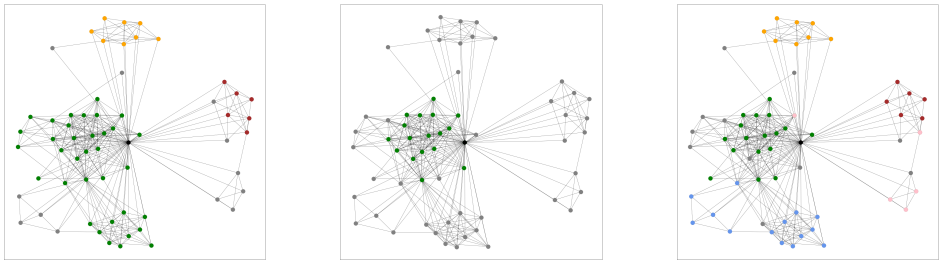
In conclusion, k-core and k-truss are both rule-based algorithms supporting query search tasks on a graph dataset. The query search problem is solved with two questions:

(1) How an index reflecting connectivity of a graph can be designed?
(2) How can the index help make query search more efficient?

For the index building part, the key is to extract the graph connectivity features: in k-core, coreness is used to describe the connectivity; in k-truss, support and trussness are proposed to measure the cohesiveness in the graph. These features are then stored in indices, which are data structures generated from the decomposition algorithm. These indices should contain the connectivity information of the original graph. They are intended to make query search problems more efficient, without losing structural information.

Both core and truss algorithms contain two steps: decomposition and query search. The general ideas of basic core decomposition method and truss decomposition algorithm are similar. They both iteratively delete the nodes with smaller coreness or trussness and reorder the nodes. Finally, they both obtain a data structure storing the coreness or trussness information. With the indices, query search algorithms are simply performing *BFS* based on connectivity information.

If core and truss decomposition algorithms applied to weighted graphs, query search problem can be extended to attribute query search problem. The distances of vector representations of nodes' attributes can be assigned to the weights of corresponding edges in the graph.



(a) K-truss (b) K-core (c) Ground-truth

Figure 7: Query result with node 698

In terms of the differences of these two algorithms, k-truss supports dynamic graph setting, the extra work to be done is to locally update edge trussness. On the other hand, it is more costly for k-core discussed in this report since the coreness has to be recomputed when the graph is modified.

Since k-core only supports mono-community search, it has limitation when the application requiring to discover multiple communities with one query node.

Besides, the connectivity measurements for community construction are also quite different. K-core utilizes the degree to construct community, while k-truss explores features on edges by using edge support to show its connectivity.

Another major difference is that the definition of cohesion and connection in k-truss allows a vertex to participate in multiple communities. Such property performs better in searching small but multiple communities than k-core in our experiments.

Future Work. In the next term, we hope to look into the graph search problem with attributes. We have went through core-based and truss-based attribute search algorithms discussed in [8, 10]. These algorithms focus on keyword type attributes. We may consider whether it is possible to extend rule-based search method to graph with high-dimensional attributes. Graph Neural Network is also taken into account to tackle this problem.

Contribution. Both members in the group contribute to this project equally. Specifically speaking, Chen Yu focuses more on k-core algorithm interpretation and Xu Xiang concentrates on k-truss part. As for experiment part, we both study the source code and program the evaluation methods together.

REFERENCES

[1] James Abello, Mauricio GC Resende, and Sandra Sudarsky. 2002. Massive quasi-clique detection. In *Latin American symposium on theoretical informatics*. Springer, 598–612.

[2] Yong-Yeol Ahn, James P Bagrow, and Sune Lehmann. 2010. Link communities reveal multiscale complexity in networks. *nature* 466, 7307 (2010), 761–764.

[3] Vladimir Batagelj and Andrej Mrvar. 1998. Pajek-program for large network analysis. *Connections* 21, 2 (1998), 47–57.

[4] Vladimir Batagelj and Matjaz Zaversnik. 2003. An O (m) algorithm for cores decomposition of networks. *arXiv preprint cs/0310049* (2003).

[5] Jonathan Cohen. 2008. Trusses: Cohesive subgraphs for social network analysis. *National security agency technical report* 16 (2008), 3–29.

[6] Wanyun Cui, Yanghua Xiao, Haixun Wang, and Wei Wang. 2014. Local search of communities in large graphs. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 991–1002.

[7] Imre Derényi, Gergely Palla, and Tamás Vicsek. 2005. Clique percolation in random networks. *Physical review letters* 94, 16 (2005), 160202.

[8] Yixiang Fang, Reynold Cheng, Siqiang Luo, and Jiafeng Hu. 2016. Effective community search for large attributed graphs. *Proceedings of the VLDB Endowment* 9, 12 (2016), 1233–1244.

[9] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying k-truss community in large and dynamic graphs. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 1311–1322.

[10] Xin Huang and Laks VS Lakshmanan. 2017. Attribute-driven community search. *Proceedings of the VLDB Endowment* 10, 9 (2017), 949–960.

[11] Jure Leskovec and Julian McAuley. 2012. Learning to discover social circles in ego networks. *Advances in neural information processing systems* 25 (2012), 539–547.

[12] Mark EJ Newman and Michelle Girvan. 2004. Finding and evaluating community structure in networks. *Physical review E* 69, 2 (2004), 026113.

[13] Gergely Palla, Imre Derényi, Illés Farkas, and Tamás Vicsek. 2005. Uncovering the overlapping community structure of complex networks in nature and society. *nature* 435, 7043 (2005), 814–818.

[14] Martin Rosvall and Carl T Bergstrom. 2008. Maps of random walks on complex networks reveal community structure. *Proceedings of the National Academy of Sciences* 105, 4 (2008), 1118–1123.

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

[16] Mauro Sozio and Aristides Gionis. 2010. The community-search problem and how to plan a successful cocktail party. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*. 939–948.

[17] Jia Wang and James Cheng. 2012. Truss decomposition in massive networks. *arXiv preprint arXiv:1205.6693* (2012).

[18] Jierui Xie, Stephen Kelley, and Boleslaw K Szymanski. 2013. Overlapping community detection in networks: The state-of-the-art and comparative study. *Acm computing surveys (csur)* 45, 4 (2013), 1–35.