MORGAN & CLAYPOOL PUBLISHERS

Community Search over Big Graphs

Xin Huang
Laks V.S. Lakshmanan
Jianliang Xu

SYNTHESIS LECTURES ON DATA MANAGEMENT

H.V. Jagadish, Series Editor

CHAPTER 3

Cohesive Community Search

This chapter discusses the problem of community search in simple graphs, and focuses on just the structural characteristics of networks. In this simplest setting, a graph represents a structure of interactions within a group of vertices. We consider an undirected, unweighted simple graph $G = (V(G), E(G))$ with $n = |V(G)|$ vertices and $m = |E(G)|$ edges. Community models in this class can only leverage the structural characteristics of networks, essentially focusing on the density of the connection structure. Given a set of query nodes, community search is to find a densely-connected subgraph containing all query nodes. Community search has attracted a great deal of attention, motivated by applications such as social circle discovery, advertising and viral marketing, content recommendation, and team formation [55]. Several different criteria to assess the goodness of a community have been proposed recently, based on dense subgraphs such as quasi-clique, k-core, k-truss, and densest-subgraph.

In the following sections, we introduce representative community models in detail. Associated with each community model is its corresponding community search problem. Specifically, we present four kinds of community search models based on quasi-clique, k-core, k-truss, and densest-subgraph respectively. Section 3.1 introduces the k-clique-based community search models, which aim at finding highly cohesive communities. However, finding k-clique [139] and even γ-quasi-k-clique [54] have been proven to be NP-hard, which imposes a severe computational bottleneck. The heuristic algorithms reduce the complexity, but cannot give a theoretical guarantee of the approximation quality [54]. Section 3.2 presents the community search models based on k-core, which has polynomial time complexity for computing the k-core subgraph and makes the k-core-based community search computationally tractable and efficient [18, 55, 122, 157]. Section 3.3 presents the community models based on k-truss, which unlike k-core considering simple edge connections only, requires each edge connection to be contained within at least $k - 2$ triangles [89, 96]. Section 3.4 studies the densest-subgraph-based community model, which uses the technique of random walk for detecting communities with highest query-biased densities [180]. For most of these community search problems, queries consisting of one or more vertices have been studied. Section 3.5 summarizes all community models introduced in this chapter.

3.1 QUASI-CLIQUE COMMUNITY MODELS

In this section, we first introduce a *clique*-based community model. Recall that a k-clique is a complete graph of k vertices where every pair of nodes is connected by an edge (see Defini-

tion 2.3.1). Then, we generalize it to a *quasi-clique*-based community model and formulate its corresponding community search problem.

3.1.1 CLIQUE-BASED COMMUNITY DETECTION

In a clique, every pair of vertices is adjacent. Clique is a widely accepted structure of communities where a group of vertices are densely connected to each other. The clique percolation method (CPM) is a well-known approach for analyzing overlapping communities in networks [139]. The high-level idea of CPM is to first find a k-clique as a seed and then expand it to a community. This approach works well on small-scale networks, but has a poor performance on large-scale networks, due to the expensive computation of k-clique enumeration.

Algorithm. The CPM method builds up the communities in a bottom-up manner. It starts from the k-cliques, i.e., complete subgraphs of k vertices. Two cliques are adjacent if they share $k-1$ vertices. Based on the definitions of k-clique and clique adjacency, a community is defined as the maximal union of k-cliques where each pair of k-clique subgraphs can be reached from each other via a series of adjacent k-cliques.

For example, given the graph G in Figure 3.1 and a parameter $k = 4$, the subgraphs of G induced by $\{v_1, v_2, v_3, v_5\}$ and $\{v_1, v_3, v_4, v_5\}$ are both k-cliques, respectively, denoted H_1 and H_2 shown in Figures 3.2a and 3.2b. The k-cliques H_1 and H_2 are adjacent since they share $(k-1) = 3$ vertices $\{v_1, v_3, v_5\}$—see Figure 3.2c. Finally, the CPM method finds one community formed by vertices $\{v_1, v_2, v_3, v_4, v_5\}$ (Figure 3.2d). The community is the union of all k-cliques (H_1 and H_2) that can be reached from each other through a series of adjacent k-cliques.

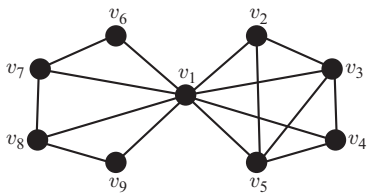
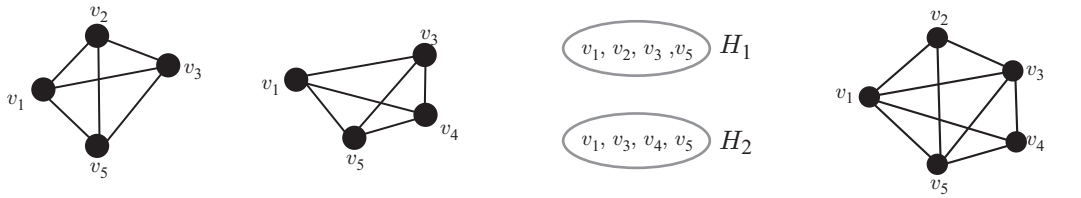


Figure 3.1: An example of graph G.

Overlapping Communities. Intuitively, given a query vertex q, there may exist several communities containing q. k-clique communities naturally form the overlapping communities of q. Figure 3.3 shows four different k-clique communities, highlighted in different colors, detected by CPM where $k = 4$ [139]. Note that two 4-cliques are adjacent if they share 3 vertices, and any k-clique can be reached only from the k-cliques of the same community through a series of adjacent k-cliques. The yellow community shares a single vertex with the blue one, whereas it overlaps with the green one in three vertices and one edge. These overlapping parts are highlighted in red.



(a) H_1 (b) H_2 (c) k-clique adjacency (d) k-clique community

Figure 3.2: An example of applying the CPM method for $k = 4$ on graph G in Figure 3.1. H_1 and H_2 are 4-cliques and share 3 vertices $\{v_1, v_3, v_5\}$, thus H_1 and H_2 are adjacent as $3 \geq (k-1)$.

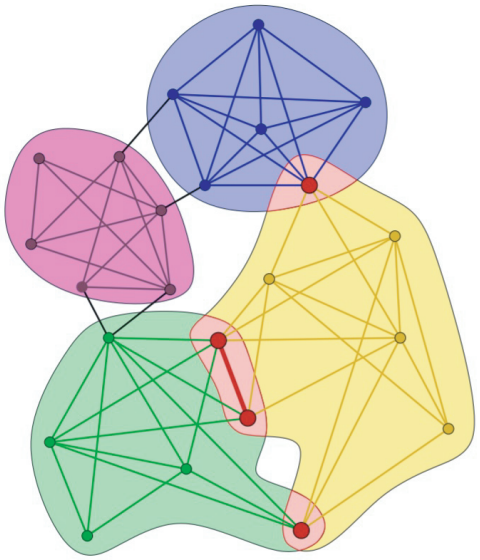
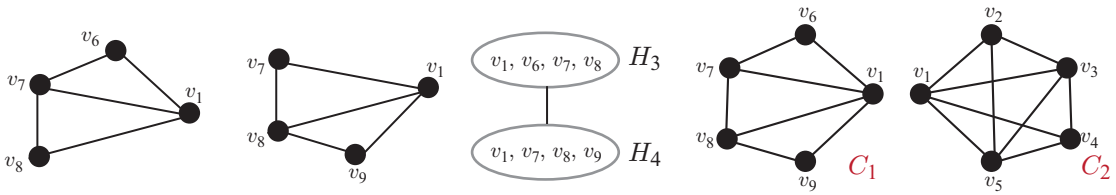


Figure 3.3: An example of k-clique communities detected by the clique percolation method (CPM) [139]. Here, $k = 4$. Used with Permission.

Limitations of clique-based community model. Intuitively, the CPM method requires to locate all maximal cliques, which is known to be an NP-hard problem. Even though this approach has already been applied successfully for analyzing networks with a few million nodes, the running time complexity is exponential in k in the worst case and it is not practical for very large values of k. Besides its high running time complexity, the definition of communities based on cliques is a restrictive notion, which limits opportunities for locating communities in real data. First, *every* pair of vertices in a k-clique must be connected, allowing no missing edges. For example, consider the graph G in Figure 3.1 and the parameter $k = 4$, graphs H_3 and H_4 shown in Figures 3.4a and 3.4b are not 4-cliques, which implies that the CPM method would miss the

community C_1 shown in Figure 3.4d. Second, two k-cliques are considered adjacent iff they share as many as $k-1$ vertices, and there is no flexibility in the amount of overlap allowed between k-cliques. This is another source of rigidity in the definition of clique-based communities.



(a) H_3 (b) H_4 (c) α-adjacency (d) α-adjacency-γ-quasi-k-cliques

Figure 3.4: An example of applying (α, γ)-OCS model on graph G in Figure 3.1. Here, $k = 4$, $\gamma = 0.8$, and $\alpha = 3$. H_1 and H_2 are γ-quasi-k-cliques, and they are α-adjacent as they share at least α vertices.

3.1.2 QUASI-CLIQUE-BASED COMMUNITY SEARCH

To address the limitations of the CPM method, Cui et al. relax its two constraints of clique and adjacency [54]. In this section, we introduce a more general model of **O**nline **C**ommunity **S**earch, called (α, γ)-OCS, based on their work.

Problem Formulation

First, the constraint of k-clique is relaxed to quasi-clique by allowing some missing edges within the subgraph. The concept of quasi-clique is one of several dense subgraphs defined in Chapter 2 (see Definition 2.3.3).

Definition 3.1.1 (γ-quasi-k-clique) *Given a positive integer number k and a number $0 \leq \gamma \leq 1$, a γ-quasi-k-clique is a graph of k vertices with at least $\lfloor \gamma \frac{k(k-1)}{2} \rfloor$ edges.*

Second, the adjacency constraint is relaxed. In (α, γ)-OCS, two γ-quasi-k-cliques are considered adjacent if they share at least α vertices, where $1 \leq \alpha \leq k-1$. It is defined as α-adjacency.

Definition 3.1.2 (α-adjacency) *Given two γ-quasi-k-cliques G_1 and G_2, they are considered adjacent if they share at least α vertices, where $1 \leq \alpha \leq k-1$.*

Based on the definitions of γ-quasi-k-clique and α-adjacency, the quasi-clique-based community is defined as follows.

Definition 3.1.3 (Quasi-clique based Community) *Given a graph G and three parameters k, α, and γ, an α-adjacency-γ-quasi-k-clique is the maximal connected subgraph such that every pair of γ-quasi-k-cliques can be reached from each other via a series of α-adjacent γ-quasi-k-cliques.*

The problem of (α, γ)-OCS is formulated as follows.

Problem 3.1.1 ((α, γ)-OCS [54]) *Given a query vertex q and three parameters k, α, and γ, the problem of (α, γ)-OCS is to find all α-adjacency-γ-quasi-k-cliques containing q.*

For example, given the graph G in Figure 3.1, consider the (α, γ)-OCS model on G with query vertex $q = v_1$, and parameters $k = 4$, $\gamma = 0.8$, and $\alpha = 3$. The subgraphs H_3, H_4 shown in Figures 3.4a and 3.4b are both γ-quasi-k-cliques, as H_3 and H_4 both have edge density as $\frac{5}{6} \geq \gamma = 0.8$. In addition, the γ-quasi-k-cliques H_3 and H_4 are adjacent as they share $(k-1) = 3$ vertices $\{v_1, v_7, v_8\}$, see Figure 3.4c. By Definition 3.1.3, H_3 and H_4 are merged into a single α-adjacency-γ-quasi-k-clique C_1 formed by vertices $\{v_1, v_6, v_7, v_8, v_9\}$; see Figure 3.4d. Moreover, it is easy to verify that subgraph C_2 shown in Figure 3.4d is also an α-adjacency-γ-quasi-k-clique containing v_1. Finally, the (α, γ)-OCS model returns two communities C_1 and C_2 as answers to the community search with the above parameters.

Query Processing Algorithms

In this section, we first introduce an exact and straightforward solution to solving (α, γ)-OCS. After that, a more efficient approximation algorithm is presented. The core idea of the efficient algorithm for finding (α, γ)-OCS is finding a γ-quasi-k-clique and meanwhile checking the adjacency constraint for pruning disqualified candidates for speed-up. Similar to CPM, this approach works well on small-scale networks, but not real large-scale networks, due to the expensive computation of γ-quasi-k-clique enumeration.

A Naive Method. The basic algorithm is outlined in Algorithm 3.4, which consists of two major stages. In the first stage, initially the query vertex q is added into a unvisited list S. Then, for each unvisited vertex, we find all γ-quasi-k-cliques that contain the vertex, and the unvisited new vertices of these γ-quasi-k-cliques are added to S. We repeat this procedure until no new vertex can be seen among the γ-quasi-k-cliques found. In the second stage, we calculate the clique adjacency for all γ-quasi-k-cliques found in the first step. Finally, we return all clique components as the resulting communities.

An Improved Algorithm. Cui et al. [54] point out that the decision version of (α, γ)-OCS problem, which determines whether there are any k-cliques containing the query vertex, is NP-hard. The naive algorithm needs to enumerate an exponential number of γ-quasi-k-cliques for a candidate community, which is clearly inefficient. Even worse, many such γ-quasi-k-cliques do not belong to a valid community. To avoid such wasteful computation, the adjacency constraint can be checked when a γ-quasi-k-clique is enumerated. Following this strategy, Cui et al. [54] propose an improved algorithm, shown in Algorithm 3.5. The algorithm runs iteratively. In each iteration it finds an unvisited γ-quasi-k-clique containing the query vertex using the procedure of *Next-Quasi-Clique*. If such a γ-quasi-k-clique exists, we *Expand* the γ-quasi-k-clique to find an α-adjacency-γ-quasi-k-cliques community by following the constraint of α-adjacency.

Algorithm 3.4 Naive (α, γ)-OCS

Input: A graph $G = (V, E)$, a query vertex q, numbers k, γ, and α.
Output: α-adjacency-γ-quasi-k-cliques containing q.

```
1: //Stage 1. Find all the candidate γ-quasi-k-cliques;
2: S ← {q}.
3: while S is not empty do
4:     for all unvisited vertex v ∈ S do
5:         mark v as visited;
6:         find all γ-quasi-k-cliques containing v;
7:         Add unvisited new vertices into S;
8: //Stage 2. Calculate clique adjacency;
9: Calculate the adjacency matrix of candidate γ-quasi-k-cliques;
10: Return α-adjacency-γ-quasi-k-cliques containing q;
```

Algorithm 3.5 Improved (α, γ)-OCS

Input: A graph $G = (V, E)$, a query vertex q, numbers k, γ, and α.
Output: α-adjacency-γ-quasi-k-cliques containing q.

```
1: R ← ∅;
2: while true do
3:     seed ← Next-Quasi-Clique(q, {q});
4:     if seed = ∅ then break;
5:     C ← Expand(seed);
6:     R ← R ∪ {C};
7: return R;
```

In the following, we detail the implementations of the procedures: *Next-Quasi-Clique* and *Expand*.

Next-Quasi-Clique.

Finding a γ-quasi-k-clique is computationally hard. A brute-force method needs to enumerate all k-sized subsets for identifying a valid γ-quasi-k-clique. Cui et al. [54] propose a procedure using depth-first search strategy with backtracking, shown in Algorithm 3.6. The method *Next-Quasi-Clique*(u, S) starts from a candidate vertex set S by exploring vertex $u's$ neighborhood. It iteratively adds a new vertex into S until an unvisited γ-quasi-k-clique G_S is found (lines 3–4). The backtracking has two significant advantages. First, it speeds up the search process by pruning impossible candidates (lines 7–9). Second, it selects a new vertex v from the neighbors of the current vertex u (lines 10–12), which ensures the answer G_S is a connected subgraph. Given a vertex set $S \subseteq V$, the maximum number of edges in a γ-quasi-k-clique that contains G_S is

$$f(S) = |E(G_S)| + \frac{(k - |S|)(k + |S| - 1)}{2}, \tag{3.1}$$

Algorithm 3.6 Next-Quasi-Clique (u, S)

Input: a vertex u, a candidate vertex set S.
Output: a γ-quasi-k-clique containing u.

```
1: ans ← ∅;
2: if |S| = k then
3:     if G_S is an unvisited γ-quasi-k-clique then
4:         return G_S;
5:     else
6:         return ans;
7: f(S) = |E(G_S)| + (k−|S|)(k+|S|−1)/2;
8: if f(S) < γ k(k−1)/2 then
9:     return ans;
10: for v ∈ N(u) − S do
11:     ans ← Next-Quasi-Clique (v, S ∪ {v});
12:     if ans ≠ ∅ then break;
13: return ans;
```

Algorithm 3.7 Expand (C)

Input: a γ-quasi-k-clique C.
Output: an α-adjacency-γ-quasi-k-clique containing C.

```
1: ans ← C;
2: for S ⊂ C, |S| = α do
3:     f(S) = |E(G_S)| + (k−|S|)(k+|S|−1)/2;
4:     if f(S) < γ k(k−1)/2 then continue;
5:     for S′ ⊂ V, |S′| = k − α do
6:         C′ ← S ∪ S′;
7:         if C′ is an unvisited γ-quasi-k-clique then
8:             ans ← ans ∪ Expand (C′);
9: return ans;
```

where $|E(G_S)|$ is the number of edges existing in G_S, and $\frac{(k-|S|)(k+|S|-1)}{2}$ results from the maximum number of candidate edges. $\frac{(k-|S|)(k+|S|-1)}{2} = (k-|S|)|S| + \frac{(k-|S|)(k-|S|-1)}{2}$ is formed by two parts: the first part is $(k-|S|)|S|$ edges between the vertices in S and the $(k-|S|)$ vertices in γ-quasi-k-clique that are not in S; the second part is $\frac{(k-|S|)(k-|S|-1)}{2}$ edges among those $(k-|S|)$ vertices. Obviously, if $f(S) < \gamma\frac{k(k-1)}{2}$, we can certainly prune the candidate S (lines 8–9).

Expand. The procedure *Expand*(C) for finding an α-adjacency-γ-quasi-k-clique containing C via α-adjacency is outlined in Algorithm 3.6. The key idea is to find a subset S of size $|S| = \alpha$ and then find another subset S' from the local neighborhood subgraph of C so that the combined subgraph $S \cup S'$ is a valid α-adjacency-γ-quasi-k-clique (lines 2–8). Note that it is not needed to

enumerate S' from the entire vertex set V (line 5). The induced graph $G_{S\cup S'}$ is always connected, thus, in the worst case, we just need to explore the $|S'|$-hop-neighborhood of S, which contains all vertices at most $|S'|$ hops away from any vertex in S [54].

Duplication Detection of γ-quasi-k-cliques. In both procedures *Next-Quasi-Clique* and *Expand*, we may generate the same γ-quasi-k-clique from different search paths. Hence, we need to identify whether a γ-quasi-k-clique has ever been generated before, i.e., visited. For this purpose, a hash table is used to store the visited γ-quasi-k-cliques, allowing for querying a visited clique in constant time. Consider the following hash function:

$$h(C) = \left(\sum_{v\in C} ID(v) \times a^{ID(v)}\right) \mod b \tag{3.2}$$

where a, b are two large primes and $ID(v)$ is the id of v. Hashing each k-sized γ-quasi-k-clique takes $O(k)$ time. Applying the DFS strategy, the complexity of such hashing computation can be further reduced to $O(1)$ [54].

Limitations of (α, γ)-OCS model. Next, we analyze the limitations of the (α, γ)-OCS model.

- First, γ as an average density measure may not necessarily guarantee a cohesive community structure. Consider the graph in Figure 3.5 which is a 0.8-quasi-7-clique containing query vertex q. However, q is only connected with one vertex in the community, thus it is obviously not a cohesive community for q.
- Second, there are three parameters α, γ, k in this model, the setting of which may vary significantly for different query vertices. For example, in a research collaboration network, the communities of a famous scholar and a junior scholar can be dramatically different in terms of the community size and density. Thus, it is difficult to choose proper values for the three parameters given an arbitrary query vertex.
- Third, finding α-adjacency-γ-quasi-k-clique has been proven to be NP-hard [54], which imposes a severe computational bottleneck. The approximate algorithms for clique enumeration and expansion proposed in [54] reduce the complexity, but they cannot give a theoretical guarantee on the approximation quality since it is NP-hard to approximate maximum cliques.

3.2 CORE-BASED COMMUNITY MODELS

In this section, we present community models built upon the dense subgraph of k-core. Recall that a k-core is the largest subgraph of graph G such that every vertex has degree at least k within this subgraph (see Definition 2.4.1). There exist several k-core-based community models such as Global-Core and Constrained-Core [157], Local-Core [55], Minimum-Core [18], and

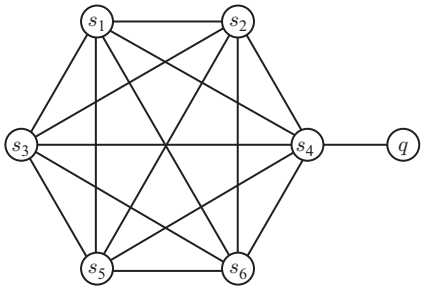


Figure 3.5: A 0.8-quasi-7-clique containing q.

k-Influential [122]. We summarize these works into three types of community models and introduce them in detail. The first one is maximum-core community model. It aims at maximizing the density of community, i.e., find a connected k-core with the largest k. The second one is minimum-sized k-core community model, which finds a k-core community with the smallest vertex size. The third one is k-Influential community model, which finds k-Influential communities with the largest influence scores.

3.2.1 MAXIMUM-CORE COMMUNITY SEARCH

Global-Core Community Search

Sozio and Gionis [157] propose one of the widely known formulations of k-core-based community search model. The problem is to find a connected subgraph that contains all query nodes and maximizes the minimum degree in this subgraph. This problem, termed Global-Core, is defined as follows.

Problem 3.2.1 (Global Core) *Given a graph $G(V, E)$ and a set of query nodes $Q \subseteq V$, find an induced subgraph $H = (V_H, E_H)$ of G, such that:*

(i) *H is a connected subgraph containing the query nodes ($Q \subseteq H$); and*

(ii) *the minimum degree of H is maximized.*

Sozio and Gionis [157] show that the Global-Core can be solved in linear time in the size of the input graph. The core idea is to adapt the peeling strategy of core decomposition by removing the weakest nodes, and finally obtaining a connected community with the largest coreness. Specifically, the algorithm applies the idea of core decomposition to remove a node having the minimum degree along with its incident edges in each iteration. The algorithm stops when the remaining connected subgraph containing the query nodes becomes disconnected. This process generates a set of immediate subgraphs. Among these subgraphs, the connected subgraph containing all query nodes and having the maximum minimum degree is returned as the answer.

Example 3.2.1 *Consider the graph G in Figure 3.6. For query vertices $Q = \{v_3, v_7\}$, the* Global-Core *community containing Q is shown in Figure 3.7. It is a connected k-*core *containing Q with the largest $k = 2$. Notice that the 3-core, shown in Figure 3.6, is disconnected, even though it contains the query nodes.*

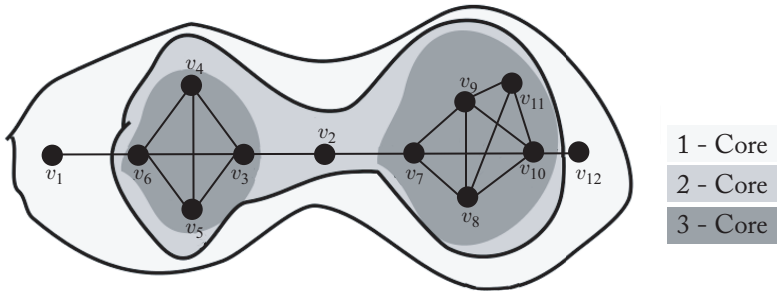


Figure 3.6: An example graph G for k-core-based community models.

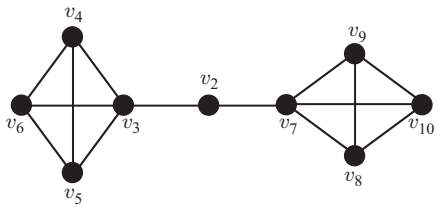


Figure 3.7: A Global-Core community for query vertices $Q = \{v_3, v_7\}$.

In addition, Global-Core tends to find quite large solutions. The large communities may involve redundant nodes or free-riders that are far away and irrelevant to the query nodes. This negatively affects the accuracy of the discovered communities. To address this issue, Sozio and Gionis [157] present a constrained version of community search problem where an upper bound on the size of the output community and an upper bound on the total distance between community members are imposed. In the following, we present the constrained version of the community search problem as Constrained-Core [157].

Constrained-Core Community Search

Two constraints, namely the size constraint and the distance constraint, are introduced in the Constrained-Core model. For the size constraint, it requires that the vertex size of a desired community H be not greater than a user-specified threshold. The distance constraint is defined as follows. First, denote by $\mathsf{dist}_G(v, q)$ the length of the shortest path between v and q in G,

where $\text{dist}_G(v, q) = +\infty$ if v and q are not connected. Given a node v in the graph G, the distance of v from the query nodes Q is defined to be $D_Q(G, v) = \sum_{q \in Q} \text{dist}_G(v, q)^2$, and $D_Q(G) = \max_{v \in V(G)}\{D_Q(G, v)\}$ is defined as the distance of the farthest node from the query nodes. For defining $D_Q(G, v)$, other alternatives are also possible, for instance, using max instead of $\sum$ or not using square.

Based on the Global-Core community model, the size constraint, and the distance constraint, the Constrained-Core community model is formulated as follows.

Problem 3.2.2 (Constrained Core) *Given a graph $G(V, E)$, a set of query nodes $Q \subseteq V$, a number d (distance constraint), and an integer s (size constraint), find an induced subgraph $H = (V_H, E_H)$ of G, such that:*

(i) *H is a connected subgraph containing the query nodes ($Q \subseteq H$);*

(ii) *$D_Q(H) \leq d$;*

(iii) *$|V_H| \leq s$; and*

(iv) *the minimum degree of H is maximized.*

Condition (i) ensures that the query nodes Q do not belong to different connected components. Condition (ii) is the distance constraint, which can avoid the pathological situations of attaching communities that are far away from the query nodes. Condition (iii) requires that H has at most s nodes. In addition, the objective function to maximize is the density measure, i.e., minimum degree in H.

Example 3.2.2 *Let us consider the graph G in Figure 3.6, and apply the definition of* Constrained-Core *community model on graph G, with query vertices $Q = \{v_3, v_7\}$, distance parameter $d = 100$, and size parameter $s = 8$. One answer of* Constrained-Core *community is shown in Figure 3.8. The number of vertices is 8, which satisfies the size constraint. The whole community is a connected k-core with the largest $k = 2$, which satisfies the distance and size constraints. It is smaller than the* Global-Core *community in Figure 3.7, as it prunes away those vertices far away from the query nodes.*

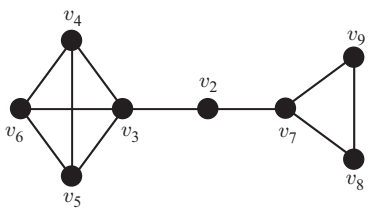


Figure 3.8: A Constrained-Core community for query vertices $Q = \{v_3, v_7\}$. Here, the distance parameter $d = 100$ and the size parameter $s = 8$.

Algorithm Greedy-Fast. A well-known combinatorial optimization problem is the *Minimum Steiner Tree* problem, defined as follows: given a graph $G = (V, E)$ with non-negative edge weights and a subset of vertices $S \subseteq V$, find a tree T in G that spans S and has the minimum total weight. T may involve vertices in $V \setminus S$. The minimum Steiner tree problem is well known to be NP-hard. From this, it follows that the problem of finding a minimum-degree-based community is also NP-hard. Thus, a heuristics method Greedy-Fast is proposed in [157] to find communities with a bounded size by achieving good quality and optimization efficiency. Specifically, Greedy-Fast performs a preprocessing phase to shrink the input graph to the s' closest nodes to the query nodes, where s' is a minimum number such that the resulting graph remains connected and contains at least s nodes. The intuition of this preprocessing phase is that the closer nodes are more likely to be the nodes that belong to their community. Next, the algorithm applies a greedy strategy to iteratively remove the nodes with the smallest degree and ensures the discovered community satisfies the distance and size constraints.

Local-Core Community Search

Cui et al. [55] study the same problem of Global-Core for the special case with a single query vertex, i.e., $|Q| = 1$.

Problem 3.2.3 (Local Core) *Given a graph $G(V, E)$ and a query vertex q, find an induced subgraph $H = (V_H, E_H)$ of G containing q, such that:*

(i) *H is a connected subgraph containing the query vertex ($q \in H$); and*

(ii) *the minimum degree of H is maximized.*

Example 3.2.3 *Consider the graph G in Figure 3.6. Applying the definition of the* Local-Core *community model on G with query vertex $q = v_7$, Figure 3.9 shows the* Local-Core *community, a connected 3-core containing q.*

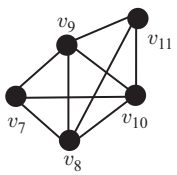


Figure 3.9: A Local-Core community for query nodes $Q = \{v_7\}$.

Algorithm. Since the methods for Global-Core and Constrained-Core both need to visit the entire input graph, their computation can be expensive over large graphs. Cui et al. [55] propose a local-search algorithm to improve the efficiency of the Global-Core algorithm. This algorithm

iteratively expands the neighborhood of the (unique) query vertex, until a subgraph that is guaranteed to contain an optimal solution has been built. Then, this subgraph is used as a reduced version of the input graph to retrieve the optimal solution. The worst-case time complexity of the Local-Core is still linear in the size of the whole input graph, but Local-Core has been shown to achieve better efficiency than Global-Core in practice. The detailed algorithm can be found in [55].

3.2.2 MINIMUM-SIZED k-CORE COMMUNITY SEARCH

Problem Formulation

Definition 3.2.1 (Minimum-Core**)** *Given a graph G and a set of query nodes Q, H is a* Minimum-Core *community, if H satisfies the following two conditions.*

(1) Connected k-core. *H is a connected k-*core *containing Q with the largest k, i.e., $Q \subseteq H \subseteq G$ and $\forall v \in V(H)$, $\deg_H(v) \geq k$.*

(2) Smallest Size. *H is a subgraph with the smallest number of vertices satisfying condition (1). That is, $\nexists H' \subseteq G$, such that $|V(H')| < |V(H)|$, and H' satisfies condition (1).*

Example 3.2.4 *Consider the graph G in Figure 3.6, and apply the definition of the* Minimum-Core *community model on G with query vertices $Q = \{v_3, v_7\}$. Figure 3.10 shows the* Minimum-Core *community for query Q. The community consists of 7 vertices and is a connected 2-core, which has minimum vertex size.*

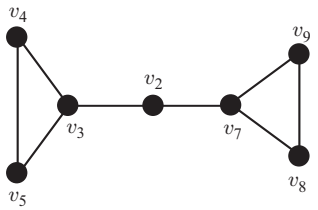


Figure 3.10: A Minimum-Core community for query vertices $Q = \{v_3, v_7\}$.

Consider the three different models Global-Core, Constrained-Core, *and* Minimum-Core. *Their query results in the above examples for the same query vertices $Q = \{v_3, v_7\}$ in graph G are, respectively, shown in Figure 3.7, Figure 3.8, and Figure 3.10. We can see that all three communities are connected 2-cores containing the query vertices Q. However, the* Constrained-Core *model is able to shrink the* Global-Core *community into a smaller one, while the* Minimum-Core *model is most stringent in finding the most compact community.*

Algorithm

Barbieri et al. [18] propose an index-based approach to solve the Minimum-Core community search effectively and efficiently. The high-level idea of Minimum-Core community search al-

gorithms is firstly constructing an index for keeping the structural information of all k-cores, and then developing an efficient heuristic strategy to connect all query nodes into a candidate community and refine it. The approach is composed of two phases: preprocessing and query processing. In the preprocessing phase, a Shell-Index is constructed in order to precompute and store some useful information for query processing. The query processing phase consists of two sub-phases: a retrieval phase, where the proper information computed/stored during the preprocessing is retrieved from the Shell-Index, and an online processing phase, where the information retrieved is further processed in order to obtain an answer to the query. In the following, we first introduce the Shell-Index.

Shell-Index. The index is constructed for the maintenance of maximal connected k-cores in graphs. The idea is to apply the core decomposition on graph G and precompute all maximal connected k-cores. Further, all maximal connected k-cores are organized into a tree-shaped structure to easily retrieve any maximal connected k-core that contains the given query vertices. This tree-shaped structure is called Shell-Index.

The data structure of Shell-Index T for graph G is defined as follows. Suppose that the maximum core number in G is $c_{\max}$. Then T has $c_{\max}$ different levels of tree nodes. We use S_i^k to denote the i-th tree node at level k of T, $1 \leq k \leq c_{\max}$. Each tree node S_i^k consists of a set of k-class vertices whose core number is k, i.e., $S_i^k \subseteq \Psi(k)$. Moreover, the subtree of T rooted at S_i^k corresponds to a maximal connected k-core in G. For example, consider the graph G in Figure 3.6. The largest core number of G is 3. It consists of three class sets $\Psi(1) = \{v_1, v_{12}\}$, $\Psi(2) = \{v_2\}$, and $\Psi(3) = \{v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}\}$. As we can see the Shell-Index T shown in Figure 3.11. T has three levels of tree nodes. At the level 1, it has a node $S_1^1 = \{v_1, v_{12}\}$ indicating vertices v_1 and v_{12} have core number 1. The subtree of T rooted at S_1^1 corresponds to a maximal connected 1-core. First, the subtree has four tree nodes and $S = S_1^1 \cup S_1^2 \cup S_1^3 \cup S_2^3 = V$, and the induced subgraph G_S is the whole graph, which is the maximal connected 1-core. Similarly, it can be easily verified that the subtrees of G rooted at S_1^3 and S_2^3 correspond, respectively, to the maximal connected 3-cores in Figure 3.6. We note that if graph G is disconnected, the Shell-Index of G will be a forest consisting of multiple trees, where each tree represents a connected component in G.

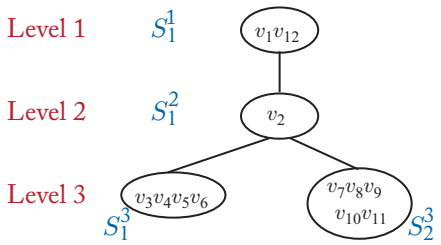


Figure 3.11: The Shell-Index for graph G in Figure 3.6.

Algorithm 3.8 Shell-Index Construction

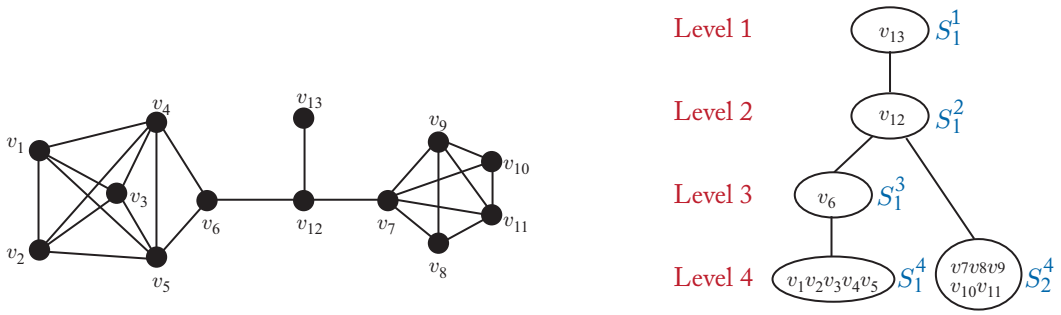
Input: A graph $G = (V, E)$.
Output: Shell-Index of G.

1: Apply the core decomposition on G using Algorithm 2.1;
2: Let $c_{\max}$ be the maximum core number in G;
3: Tree-shaped structure $T \leftarrow \emptyset$;
4: Graph $H \leftarrow \emptyset$;
5: **for** $k \leftarrow c_{\max}$ to 1 **do**
6: Let k-class set $\Psi(k) = \{v : v \in V, \varphi(v) = k\}$.
7: Let k-edge set $E_k = \{(v, u) \in E : \varphi(v) \geq k, \varphi(u) \geq k, \min\{\varphi(v), \varphi(u)\} = k\}$;
8: Adding the subgraph $H'(\Psi(k), E_k)$ into H to find all maximal connected k-cores $\{C_1, \ldots, C_r\}$ using Union-Find forest;
9: **for** $i \leftarrow 1$ to r **do**
10: Add a tree node S_i^k where $S_i^k = \Psi(k) \cap C_i$, denoted by the i-th node at the k-th level of Tree T;
11: **if** $\exists S_j^{k'} \subseteq C_i$ where $k < k' \leq c_{\max}$ **and** $S_j^{k'}$ has no parent in T **do**
12: Add a relationship <parent, child> between nodes S_i^k and $S_j^{k'}$ in tree T.
13: **return** tree-shaped structure$T$ as Shell-Index of G;

Shell-Index **Construction.** Algorithm 3.8 presents a method of constructing Shell-Index for a graph G. The basic idea is to compute all maximal connected components and then organize them into a tree-shaped structure, based on an inclusion rule that the $(k + 1)$-core is a subgraph of the k-core, for all k. However, for each vertex v with core number $\varphi(v) = l$, v may be present in many maximal connected k-cores for $1 \leq k \leq l$. To avoid duplicate information and save storage cost, each vertex v is stored only once to indicate the maximal connected k-core with the largest k, that contains v. In this way, Shell-Index is organized in an elegant tree structure. The algorithm starts by applying the core decomposition on G using Algorithm 2.1 (line 1). Then, we obtain all core numbers of vertices and let $c_{\max}$ be the maximum one (line 2). We construct the Shell-Index T from scratch level-by-level in a bottom-up manner (lines 3–12). In other words, we first create the k-th-level tree nodes and then create the $(k - 1)$-th-level tree nodes. At the level k (lines 6–12), we define a set of nodes with core number k, as $\Psi(k) = \{v : v \in V, \varphi(v) = k\}$ and the k-edge set $E_k = \{(v, u) \in E : \varphi(v) \geq k, \varphi(u) \geq k, \min\{\varphi(v), \varphi(u)\} = k\}$ (lines 6–7). We add the subgraph $H'(\Psi(k), E_k)$ into H to find all maximal connected k-cores $\{C_1, \ldots, C_r\}$ (line 8). Then, for each maximal connected k-core C_i, we create a subtree of T rooted by S_i^k (lines 9–12). The tree nodes S_i^k are formed by a set of k-class vertices in C_i, i.e., $S_i^k = \Psi(k) \cap C_i$. Then, we add tree edges between S_i^k and $S_j^{k'}$ where $k' > k$, indicating that each vertex present in the subtree of T rooted at $S_j^{k'}$ also belongs to the maximal connected k-core C_i. Finally, the algorithm returns T as Shell-Index (line 13).

Example 3.2.5 *Figure 3.12 shows a special case of tree edges that are between S_i^k and $S_j^{k'}$, where $k' > k + 1$. Consider the graph G in Figure 3.12a, and the corresponding* Shell-Index *of graph G is*

shown in Figure 3.12b. The induced subgraph of G by the vertex set $\{v_7, v_8, v_9, v_{10}, v_{11}\}$ *is a connected 4-core, thus vertices* $\{v_7, v_8, v_9, v_{10}, v_{11}\}$ *are stored in the tree node* S_2^4 *at the level-4 of* Shell-Index. *The vertex* v_{12} *can present in a connected 2-core, but does not belong to any 3-core. Thus,* v_{12} *is stored in the tree node* S_1^2 *at the level 2 of* Shell-Index. *Due to the edge* (v_{12}, v_7), *we add a tree edge between* S_1^2 *and* S_2^4, *indicating that a maximal connected 4-core rooted at* S_2^4 *belongs to a maximal connected 2-core rooted at* S_1^2.



(a) Graph G (b) The Shell-Index of graph G

Figure 3.12: An example of Shell-Index.

Now, based on the Shell-Index, we are ready to present the query processing phase in detail. This phase consists of two steps: *retrieval* of the information computed from Shell-Index, and *online processing* of this information to obtain the final query answer.

Retrieval from Shell-Index**.** Given a set of query vertices Q, the retrieval phase aims at finding a maximal connected k-core H^* with the largest k containing all query vertices Q. To retrieve H^* from Shell-Index, the problem is turned into one of finding a subtree of T rooted at a k-th level node with the largest k. This is a classical problem of finding the lowest-common-ancestor (LCA). In the worst case, for every vertex $q \in Q$, we may visit each tree node S containing q all the way to the root, and finally find the lowest-common-ancestor. The time complexity is $O(|Q|c_{\max} + |H^*|)$. The well-known Tarjan's offline lowest common ancestors (LCA) algorithm can also be applied to store Shell-Index, which enables constant time online retrieval [69].

Online processing. The retrieval phase described above finds the set H^* containing all solutions to Global-Core for a given query Q. The goal of the online processing phase is to further refine H^* to extract a solution that is as small as possible. Since the Minimum-Core community search problem is NP-hard, Barbieri et al. [18] propose an efficient heuristic algorithm called Greedy-Connection. The idea of Greedy-Connection is as follows. A solution to Minimum-Core needs to satisfy two constraints: (a) the query vertices are connected, and (b) the solution has minimum degree no less than the core number of any vertex in H^*. Among all possible solutions satisfying these constraints, the goal is to output a community with the smallest vertex size. The main

intuition of Minimum-Core is to look at constraints (a) and (b) one by one. In particular, one may first find a solution that satisfies constraint (a) only, and then refine this solution in order to make it satisfy constraint (b) too. The motivation is that in real-world graphs with power-law-like degree distribution, the minimum degree of a community is typically small (i.e., in the order of a few tens or even less); thus, it is likely that any solution that satisfies (only) constraint (a) needs just a very few additional vertices to satisfy constraint (b) too. As a result, Barbieri et al. [18] apply a bottom-up method to find a connected subgraph using as few more vertices as possible. After that, they perform a sequential procedure that first ensures connection among the query vertices, and then aims at satisfying the constraint on the minimum degree. They apply the Steiner Tree algorithm [108] to connect all query vertices Q.

3.2.3 INFLUENTIAL COMMUNITY SEARCH

Motivation. In all previous studies on community detection and search, a community is defined as a densely connected subgraph. This ignores another important aspect, namely the "influence" (or importance) of a community. In many applications, we are interested in finding the most influential communities. For example, consider the following two scenarios. Suppose that Alice is a database researcher. She may want to identify the most influential research groups from the co-authorship network of the database community, so as to be aware of the recent trends of database research by the "movers and shakers," which can be modeled using those influential groups and following their publications and blogs. Another example is the influential communities in the social network. A user Bob may intend to follow the most influential groups of three different topics "technology," "investments," and "politics," so that he can track the recent activities from those three influential groups and may conduct further analytics on that activity data. Both of the above applications motivate the identification of the most influential communities within a network [122].

Note that [122] proposes to measure the influence of a community by counting the minimum value of influential importance among all nodes in the community. The node influence is different from influential edges in social networks, which are based on independent cascade (IC) model or linear threshold (LT) model [28, 42, 75, 105, 127]. Several studies of influential community search [120, 184] have recently been conducted using diffusion models, which leave the study of influential community search under diffusion models such as IC and LT models as an open problem.

Problem Formulation

In this section, we describe k-Influential community search, the problem of finding influential communities in large networks. For this, Li et al. [122] propose a new community model called k-Influential community based on the well-known concept of connected k-core, where the importance of nodes is taken into account. To find the k-Influential community, the key ideas of the proposed solutions are building an index that incorporates the importance of nodes and

the structure of k-core, and developing an index-based online query processing algorithm for quickly identifying the k-Influential community containing query nodes. In the following, we introduce the problem setting using several new definitions.

Graph with node weights. Consider an undirected graph $G = (V, E)$ and a node weight function w, such that each node v in G is associated with a non-negative weight $w(v)$, denoting the influence (or importance) of the node v. Such weight can be computed using the PageRank algorithm [138], centrality, degree, expected spread computed using a standard diffusion model [105], or other user-defined ranking functions. Additionally, without loss of generality, we assume that the weight of each node is distinct. Note that if that is not the case, we can break ties using node identity whenever $w(v_i) = w(v_j)$, for distinct nodes v_i and v_j [122]. The influence score of a subgraph is defined as follows.

Definition 3.2.2 (Influence Score) *Given a subgraph H of graph G, the* influence score *of H is the minimum weight of the vertices in H, i.e., $f(H) = \min_{v \in V_H} w(v)$.*

Example 3.2.6 *Consider the weighted graph G in Figure 3.13 where each vertex has a weight, shown in red color, e.g., vertex v_6 has weight 6. The subgraph $H_1 \subseteq G$ shown in Figure 3.14a has weight $f(H_1) = \min_{v \in \{v_5, v_7\}} w(v) = 5$. Similarly, the subgraphs H_2 and H_3 in Figures 3.14b and 3.14c both have weight $f(H_2) = f(H_3) = 5$.*

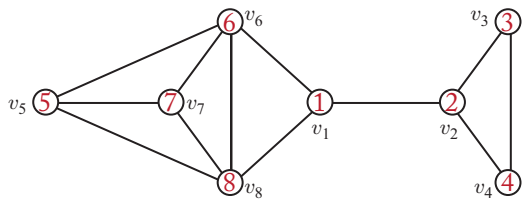
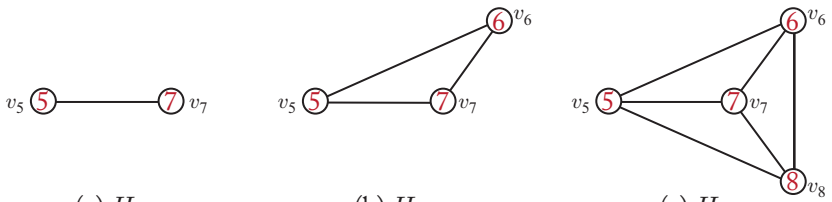


Figure 3.13: An example of weighted graph G (the numbers in red denote the node weights).



(a) H_1 (b) H_2 (c) H_3

Figure 3.14: An example of k-Influential community H_3 where $k = 2$. H_1 and H_2 are not k-Influential communities for any $k \in \{1, 2, 3\}$.

Based on the definitions of k-core and influence score, a k-Influential community is defined as follows.

Definition 3.2.3 (K-Influential Community) *Given a graph G and a parameter k, the k-Influential community is defined as a maximal connected k-core H such that there exists no other connected k-core H' having $f(H') = f(H)$ and $H \subset H'$.*

In the k-Influential community model, the parameter k measures the cohesiveness of the community. Note that a maximal connected k-core H must be an induced subgraph of G, thus the requirement of induced subgraph is redundant and is removed from the definition of k-Influential community [122].

Example 3.2.7 *Consider the graph G in Figure 3.13 and $k = 2$. The subgraph H_3 is a k-Influential community, since it is a maximal connected 2-core such that each vertex in H_3 has degree at least 2 and $f(H_3) = 5$. On the other hand, H_1 and H_2 are not 2-influential communities. H_1 is not even a connected 2-core, since each vertex has degree only 1. On the other hand, although H_2 is a connected 2-core, H_2 violates the maximal property (see Definition 3.2.3). H_2 is not a maximal connected 2-core with $f(H_2) = 5$, since $H_2 \subset H_3$ and $f(H_2) = f(H_3) = 5$. Notice that H_3 is also a maximal connected 3-core with influence score $f(H_3) = 5$, and is thus also a k-Influential community for $k = 3$.*

Intuitively, a good influential community should not only be a strongly cohesive induced subgraph, but also have a large influence value. In many practical applications, we are typically interested in the most influential communities whose influence values are larger than those of other influential communities. Based on these motivations, the definition of k-Influential community search is given as follows.

Problem 3.2.4 (k-Influential) *Given a graph $G = (V, E)$, a weight function w, and parameters k and r, find the top-r k-Influential communities with the highest influence scores.*

Example 3.2.8 *Consider the graph G in Figure 3.13 and parameters $k = 3$ and $r = 1$. The solution for k-Influential on G is the k-Influential community H_3 in Figure 3.14c, since H_3 has the highest influence score $f(H_3) = 5$. However, if we modify the parameters to $k = 2$ and $r = 1$, then the solution will be the subgraph of G by induced by the vertices $\{v_6, v_7, v_8\}$, since $\triangle_{v_6v_7v_8}$ is the 2-core with the highest influence score $f(\triangle_{v_6v_7v_8}) = 6$.*

It is important to note that although we define the influence score of a community as the minimum weight of a node in the community, we could also in principle aggregate the weights in other ways, e.g., MAX, by defining the influence score as the maximum weight of the nodes in the community. The techniques proposed in [122] (e.g., ICP-Index to be introduced in Section 3.2.3) can be easily extended to process queries for the MAX aggregation of influence scores. In place of minimum weight, we could aggregate node weights using other aggregate functions such as MAX or AVG in order to define the influence score of a community. In this case, some simple modifications to the proposed techniques are needed to solve this more general k-Influential community search problem.

DFS-Based Query Processing Algorithm

We first make some observations of k-Influential communities from the example graph G in Figure 3.13.

Observation 3.1 Given a graph G and any k-core H of G, each maximal component of H is a k-Influential community. For example, the 2-core of graph G in Figure 3.13 is itself; G is the maximal component of G, and is thus a 2-Influential community. Its influence score is 1.

Observation 3.2 For any k-Influential community H, let v be the vertex that has the smallest weight in H, i.e., $v = \arg\min_{v \in V(H)} w(v)$. Let us first remove the vertex v and all its incident edges from H. Since the remaining graph may have vertices with degree less than k, we then continue removing these unqualified vertices and their incident edges until the remaining graph is empty or all remaining vertices have degree at least k in H. As a result, each maximal connected component of the remaining graph H is a k-Influential community. For example, consider the graph G from Figure 3.13. As seen above, it is a 2-influential community, and the vertex v_1 has the smallest weight of 1. Let us remove the vertex v_1 and its incident edges from G. The remaining graph satisfies the k-core property and contains two maximal connected components R_2 and R_3 (see Figures 3.15b and 3.15c), which are k-Influential communities.

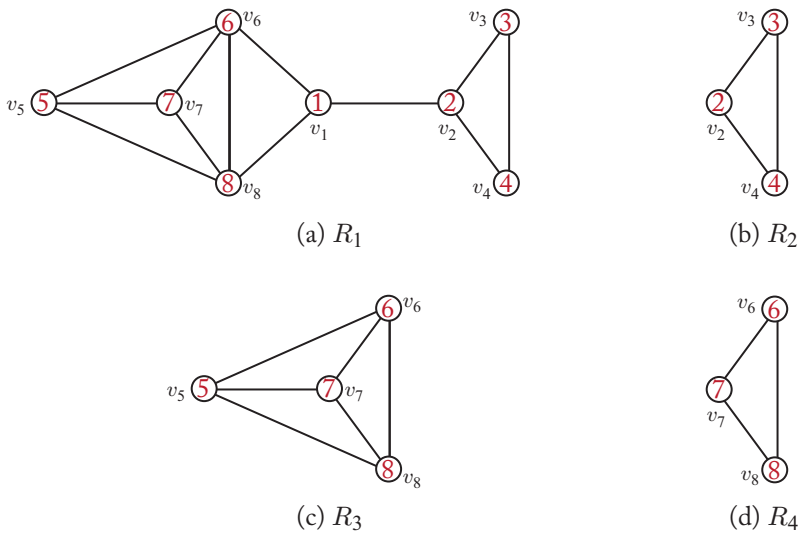


Figure 3.15: All k-Influential communities $\{R_1, R_2, R_3, R_4\}$ of graph G in Figure 3.13. Here, $k = 2$.

Algorithm 3.9 The DFS-based algorithm

Input: A graph $G = (V, E)$, a weight function w, and parameters k and r.
Output: The top-r k-Influential communities.

```
1: Find k-core H of G using the core decomposition in Algorithm 2.1;
2: l ← 0;
3: while H ≠ ∅ then
4:     l ← l + 1;
5:     v* ← arg max_{v∈V(H)} w(v);
6:     Let H_{v*} be the maximal connected component of H containing v*;
7:     R_l ← H_{v*}; // H_{v*} is a k-Influential community;
8:     Delete v* and its incident edges from H;
9:     Maintain k-core property of H by iteratively removing vertex v with deg_H(v) < k and its incident edges;
10: if l ≥ r then
11:     return {R_l, ..., R_{l−r+1}};
12: else
13:     return {R_l, ..., R_1};
```

Algorithm. Based on the above observations, we are ready to present an efficient algorithm for k-Influential community search in Algorithm 3.9. The key idea is largely similar to that for k-core decomposition. First, for the given parameter k, we find the k-core H of graph G using the core decomposition in Algorithm 2.1 (line 1). Then, we iteratively invoke the following procedure until the resulting graph H becomes empty (lines 3–9). The procedure has two steps. The first step is to find a vertex v^* with the smallest weight in H. Based on Observation 3.1, the maximal connected component of H containing v^*, denoted H_{v^*}, is a k-Influential community. We have the influence score $f(H_{v^*}) = w(v^*)$ and add H_{v^*} to our list of k-Influential communities (lines 4–7). The second step is to remove $v*$ and its incident edges from H and maintain k-core property in the remaining graph H, i.e., each vertex in H has degree at least k (lines 8–9). Finally, if the number of discovered k-Influential communities is less than r, we return all k-Influential communities the algorithm found; otherwise, we return the top-r k-Influential communities $\{R_l, \ldots, R_{l-r+1}\}$ with the highest influence scores.

Example 3.2.9 *We apply Algorithm 3.9 on the graph G in Figure 3.13 with parameters $k = 2$ and $r = 1$. The whole graph G is 2-core, H is identical to G (line 1). In the first iteration, we identify the vertex $v^* = v_1$ with the smallest weight of 1 and obtain the k-Influential community $R_1 = G$ in Figure 3.15a. After the removal of vertex v_1, the remaining graph consists of two components as shown in Figures 3.15b and 3.15c. In the second iteration, we identify the vertex $v^* = v_2$ and obtain the maximal connected component $H_{v^*} = R_2$ in Figure 3.15b. After removing v_2 from the graph, vertices v_3 and v_4 will be also deleted for violating the property of k-core. Repeating the above process, in the third and fourth iterations, k-Influential communities R_3 and R_4 in Figures 3.15c and 3.15d are obtained. Finally, for $r = 1$, the algorithm returns $\{R_1\}$ as the (top-r) answer.*

Complexity Analysis. The time complexity and space complexity is $O(m+n)$, which is the same as the complexity of core decomposition algorithm.

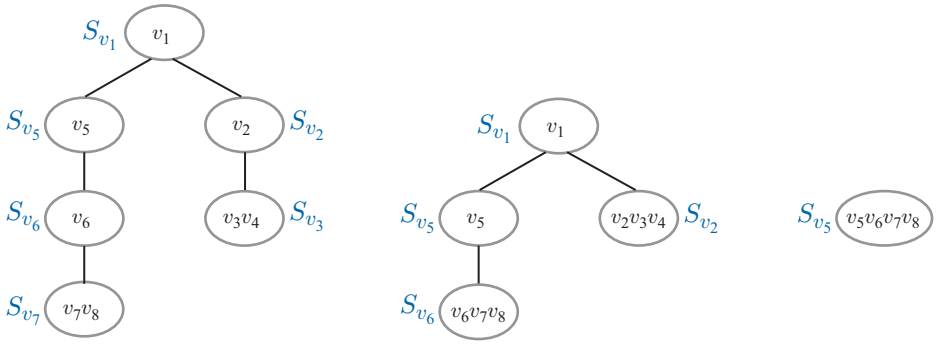
ICP-Index-Based Query Processing Algorithm

Although Algorithm 3.9 takes linear time in graph size, an index-based algorithm can further improve efficiency for query processing. The basic idea is to first precompute all k-Influential communities for every k, then use a space-efficient structure called ICP-Index to keep track of all k-Influential communities in memory. Based on ICP-Index, the algorithm produces all top-r results in optimal time.

A novel ICP-Index. We first introduce ICP-Index using the Example 3.2.9. Assume $k = 2$ and consider all k-Influential communities $\{R_1, R_2, R_3, R_4\}$ of graph G, as shown in Figure 3.13. The k-Influential communities R_1, R_2, R_3, R_4 have increasing influence scores. Moreover, R_2 and R_3 are subgraphs of R_1 and recursively R_4 is a subgraph of R_3. We define an inclusion relationship on k-Influential communities as follows. Given two k-Influential communities A and B with $A \subset B$, we say A is a sub-k-Influential community of B. Based on such an inclusion relationship, all k-Influential communities can be organized in a tree-shaped (more generally, a forest-shaped) structure. However, instead of storing all communities explicitly, which is expensive, we can represent them compactly. We only store those nodes of a k-Influential community H that are not included in any sub-k-Influential communities of H. For example, in the k-Influential community R_1, the vertex v_1 does not belong to any sub-k-Influential communities R_2, R_3, or R_4. Thus, in the ICP-Index in Figure 3.16b (for $k = 2$), we create an isolated tree node $S_{v_1} = \{v_1\}$ containing v_1. It can be seen in a similar way that the node corresponding to S_{v_5} in Figure 3.16b only needs to store the vertex v_5. The tree of ICP-Index rooted at S_{v_1} corresponds to the k-Influential community R_1, i.e., the union of the vertex sets of all tree nodes in Figure 3.16b is $V = S_{v_1} \cup S_{v_2} \cup S_{v_5} \cup S_{v_6}$. Similarly, the tree of ICP-Index rooted by S_{v_5} corresponds to the k-Influential community R_3.

ICP-Index Construction. The ICP-Index construction algorithm is outlined in Algorithm 3.10. The general idea is to repeatedly run Algorithm 3.9 for finding all k-Influential communities for any k. More precisely, it consists of two steps: *generating tree nodes* and *generating tree edges*. First, it invokes Algorithm 3.9 $c_{\max}$ times, where $c_{\max}$ is the maximum core number in G. That is, for each $1 \le k \le c_{\max}$, it applies Algorithm 3.9 to generate isolated tree nodes in the tree T_k for k-Influential communities. Second, it invokes a procedure of tree construction that adds edges between tree nodes generated in Step 1, in order to build ICP-Index.

Generating Tree Nodes. In the procedure of tree node generation, after the deletion of v^*, all vertices removed because of violation of the k-core property in H must be stored in a tree node (lines 10–13 of Algorithm 3.10). Notice that the vertex v^* with the smallest weight is also included in the tree node S_{v^*}. The reason is that all these deleted nodes are excluded in any sub-k-Influential communities $H' \subset H$. For example, consider the second iteration of Al-



(a) ICP-index ($k = 1$) (b) ICP-index ($k = 2$) (c) ICP-index ($k = 3$)

Figure 3.16: Tree organization of all the k-Influential communities of graph G in Figure 3.13.

gorithm 3.10 for $k = 2$ and vertex $v^* = v_2$. The corresponding tree node is $S_{v_2} = \{v_2, v_3, v_4\}$. After removal of vertex v_2, vertices v_3 and v_4 are deleted due to the violation of the k-core property.

Generating Tree Edges. After generating all tree nodes in the previous step, the procedure ConstructTree adds the edges by connecting tree nodes for each tree T_k, $1 \leq k \leq c_{\max}$. First, we treat each isolated node of T_k as a single-node tree. Then, we iteratively "merge" two trees into one and finally obtain the tree (or forest) structure as ICP-Index. Here the merge operation between two trees P_1 and P_2 is defined as follows. Let S_1 and S_2 be the roots of subtrees P_1 and P_2, respectively. Assume that $f(S_1) < f(S_2)$ where $f(S_i) = \min_{u \in S_i} w(u)$ for $i = 1, 2$. Then, the merge operation between S_1 and S_2 is to add an edge between S_1 and S_2 indicating S_2 is a child node of S_1. We note that such a bottom-up tree construction algorithm for all k can be done by traversing the graph once, in decreasing order of node weights (lines 3–13).

Query Processing Algorithm. Based on the ICP-Index, the query processing algorithm is straightforward. To find the top-r k-Influential communities with the highest influence scores for a given k, we first identify the tree T_k in the ICP-Index. Given any tree node $S \in T_k$, denote the subtree of T_k rooted at S as T_k^S, which corresponds to a k-Influential community. In other words, let the vertex set be $C = \cup_{x \in T_k^S} x$, then the induced subgraph G_C is a k-Influential community. Thus, we then generate answers by returning r subtrees T_k^S with the highest weights, where the weight of a tree node is the minimum weight of nodes in its root vertex, i.e., $f(T_k^S) = \min_{x \in T_k^S, v \in x} w(v)$. The time complexity of the ICP-Index based query processing algorithm is linear in the size of the answer, which is optimal.

Algorithm 3.10 ICP-Index Construction

Input: A graph $G = (V, E)$ and a weight function w.
Output: ICP-Index of G.

Apply the core decomposition on G using Algorithm 2.1;
Let $c_{\max}$ be the maximum core number in G;
for $k \leftarrow 1$ to $c_{\max}$ **do**
  Compute k-core H of G based on the core numbers of vertices;
  $T_k \leftarrow \emptyset$; $l \leftarrow 0$;
  while $H \neq \emptyset$ **then**
    $l \leftarrow l + 1$;
    $v^* \leftarrow \arg\max_{v \in V(H)} w(v)$;
    Let H_{v^*} be the maximal connected component of H containing v^*;
    Delete v^* and its incident edges from H;
    Maintain k-core property of H by iteratively removing vertex v with $\deg_H(v) < k$ and its incident edges;
    Let S_{v^*} be the set of vertices removed from H at this loop and $v^* \in S$;
    Add an isolated node S_{v^*} into tree T_k;
return ConstructTree();

Procedure ConstructTree()
// This procedure adds the connecting edge between vertices in each tree T_k;
for all node u in G sorted in decreasing order of $w(u)$ **do**
  for $\forall v \in N(u)$ with $w(v) > w(u)$ **do**
    for $i \leftarrow 1$ **to** $\min c_u, c_v$ **do**
      $S_u \leftarrow$ the root of a tree containing u in T_i;
      $S_v \leftarrow$ the root of a tree containing v in T_i;
      if $S_u \neq S_v$ **then**
        Add an edge between S_u and S_v indicating that S_v is a child of S_u;
return $\{T_1, ..., T_{c_{\max}}\}$;

Discussion

Li et al. [122] study another problem formulation of k-Influential using a *non-containment* constraint. It is defined as follows. Given a k-Influential community H, H satisfies the non-containment constraint if and only if there exists no k-Influential community $H' \subset H$ such that $f(H') > f(H)$ holds. Thus, H is called a *non-contained* k-Influential *community*. The problem of non-contained k-Influential is to find the top-r non-contained k-Influential communities with the highest influence scores [122]. Since there is no inclusion relationship among the top-r non-contained k-Influential communities, no redundant results are included. To solve the problem of non-contained k-Influential, a slight modification needs to be made to Algorithm 3.9, by adding one more check. For a k-Influential community H, we conclude H is a non-contained k-Influential community provided H does not contain a k-core, upon removing the node with the smallest weight from H.

Table 3.1: Comparison of state-of-the-art k-core-based methods [18]. The ratings of the first four methods Global-Core, Constrained-Core, Local-Core, and Minimum-Core for Empirical Efficiency and Quality are from [18]. k-Influential is rated by us, based on the linear running time of graph size and high quality of specific communities with the maximum influence score.

Methods	Empirical Efficiency	Quality	Query Vertices	Parameter-Free
Global-Core [157]	+	+	Multiple	Yes
Constrained-Core [157]	+	++	Multiple	No
Local-Core [55]	++	++	Single	Yes
Minimum-Core [18]	+++	+++	Yes	Yes
k-Influential [122]	+++	+++	No	No

3.2.4 COMPARISON OF VARIOUS k-CORE COMMUNITY MODELS

A comparison of the state-of-the-art k-core-based methods is provided in Table 3.1, which is extended from the comparison table in [18] by additionally including k-Influential in the comparison. We compare all methods in terms of empirical efficiency, quality, the number of query vertices, and whether they are parameter-free.

In terms of empirical efficiency, the method of Constrained-Core is the worst. The problem of Constrained-Core is shown to be NP-hard, and Sozio and Gionis [157] devise heuristics that perform even worse than the known heuristics for standard Global-Core. Besides the efficiency limitations, the available algorithms for Constrained-Core also suffer from providing no guarantee w.r.t. the optimal minimum degree. The Local-Core method deals with the same problem as Global-Core for the special case of a single query vertex. In terms of improving efficiency and quality, Local-Core is proposed for searching for communities in the local graph structure and speeds up the search processing by avoiding global search.

Another problem proposed in [55] and [18] is to find a k-core with the smallest size that contains a set of given query vertices Q. Cui et al. [55] define this problem for the case of a single query vertex, i.e., $|Q| = 1$, and they do not propose algorithms. They show that this problem is NP-hard. Barbieri et al. [18] extend this result to the general case where $|Q| \geq 1$ by using a reduction from the minimum Steiner tree problem and propose a general approach called Minimum-Core to find k-core-based communities containing multiple query vertices. This method improves the efficiency of the above methods and also offers a method for finding Local-Core proposed by Cui et al. for the special case of a single query vertex. Moreover, k-Influential method address a different problem from the above. The time complexity of ICP-Index based query processing algorithm is optimal w.r.t. the answer size.

3.3 TRUSS-BASED COMMUNITY MODELS

Motivation. In social networks, it is quite typical that pairs of friends have several common friends, thus forming many triangles [60]. Indeed, triangles are regarded as the fundamental building blocks of networks and lead to a high clustering coefficient [20, 136, 165, 174, 177]. In a social network, a triangle indicates that two friends have a common friend "endorsing" their friendship, which shows a strong and stable relationship among the three friends. Intuitively, the more common friends two people have, the stronger their relationship.

In this section, we discuss community models based on the dense subgraphs of k-truss. Given a graph G, the k-truss of G is the largest subgraph in which every edge is contained in at least $(k-2)$ triangles within the subgraph [50] (see Definition 2.4.4). The k-truss is a type of cohesive subgraph defined based on triangles which model stable relationships among three nodes. However, the k-truss subgraph may be disconnected, for example, the subgraph shown in Figure 3.17a, consisting of the two shaded regions, forms the 4-truss which is obviously disconnected. Thus, the classical k-truss subgraph may not correspond to a meaningful community. To address this issue, Huang et al. [89, 96] propose two different constraints to build up densely-connected community models. One truss community model is based on the triangle connectivity such that every pair of edges of a truss community should be connected to each other via a series of triangles. The high-level idea of finding triangle-connected communities is to build an index, incorporating the triangle connectivity and k-truss structure. An index-based truss community search algorithm is proposed to find answers in optimal time. Another truss community model is based on the k-truss with the smallest diameter. The core idea of closest community search is to find a k-truss connecting all query nodes and then shrink the community to reduce the diameter as much as possible. In the following, we will introduce in detail these two truss community models, based on (i) triangle connectivity and (ii) diameter.

3.3.1 TRIANGLE-CONNECTED TRUSS COMMUNITY SEARCH

An *triangle connectivity* constraint is imposed on top of the k-truss, that is, any two edges in a community either belong to the same triangle, or are reachable from each other through a series of adjacent triangles. Here two triangles are said to be *adjacent* if they share a common edge. The triangle connectivity requirement ensures that a discovered community is connected and cohesive. This defines *triangle-connected truss community model*.

Notions and Notations

In the following, we give a series of definitions and then formulate the model. A triangle in G is a cycle of length 3. Let $u, v, w \in V$ be the three vertices on the cycle. We denote this triangle by $\triangle_{uvw}$. Then the *support* of an edge is defined as follows.

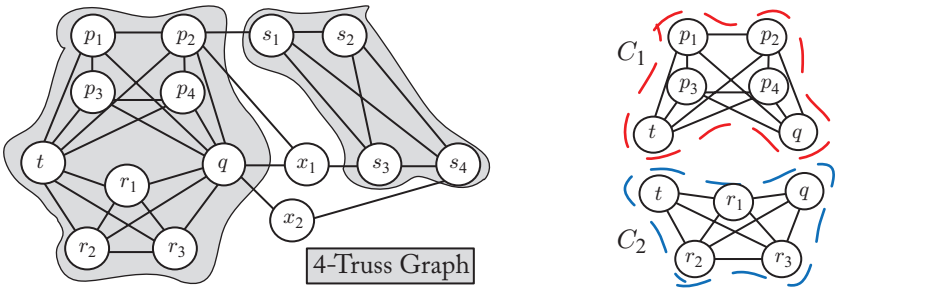
Definition 3.3.1 (Support) *The support of an edge $e(u,v) \in E$ in G, denoted $sup(e,G)$, is defined as $|\{\triangle_{uvw} : w \in V\}|$. When the context is obvious, we replace $sup(e,G)$ by $sup(e)$.*

We write $e \in \triangle$ to indicate that the edge e belongs to the triangle $\triangle$. We next define *triangle adjacency* and *triangle connectivity*.

Definition 3.3.2 (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 3.3.3 (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 $1 \leq i < n$, $\triangle_i \cap \triangle_{i+1} \neq \emptyset$.*

For the graph G in Figure 3.17a, $e(q, p_4)$ is contained in $\triangle_{qp_3p_4}$ and $\triangle_{qp_2p_4}$, thus its support $sup(e(q, p_4)) = 2$. $\triangle_{qp_3p_4}$ and $\triangle_{qp_2p_4}$ are triangle adjacent as they share a common edge $e(q, p_4)$. $\triangle_{tp_3p_4}$ and $\triangle_{qp_2p_4}$ are triangle connected through $\triangle_{qp_3p_4}$ in G.



(a) Graph G (b) Two 4-truss communities for q

Figure 3.17: k-truss community example.

Problem Formulation

Based on the definitions of support and triangle connectivity, we define the model of triangle connected truss community in the following. For simplicity, we also call the *triangle connected k-truss community* as *k-truss community* for short, and use both of them interchangeably in this book.

Definition 3.3.4 (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) Triangle 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).*

Actually the largest subgraph that satisfies condition (1) is exactly the k-truss definition used in the literature [50, 165]. However, the k-truss condition itself is insufficient to define a cohesive and meaningful community due to the following two reasons. First, a k-truss subgraph can be disconnected, thus does not represent a cohesive community. For example, as seen earlier, in Figure 3.17a, the subgraph consisting of the two shaded regions is the 4-truss, which is disconnected. So this 4-truss subgraph does not correspond to a meaningful community. Second, for a fixed k value, any vertex can belong to at most one k-truss subgraph. This limitation cannot deal with a common scenario that a user can participate in multiple communities.

With these considerations, the triangle connectivity requirement is imposed in condition (2) to ensure that the discovered communities are connected and cohesive. The rationale is that a triangle represents a strong and stable relationship among three vertices. If any two edges in a subgraph are reachable from each other through a series of adjacent triangles, the subgraph must be connected, and have a cohesive structure among all involved vertices. This definition also allows a vertex to participate in multiple communities.

Example 3.3.1 *Two 4-truss communities containing vertex q are shown in Figure 3.17b as C_1 and C_2. We can verify that every edge in C_1 is contained in at least two triangles, any two edges in C_1 are reachable through adjacent triangles, and C_1 is maximal. Thus, C_1 is a 4-truss community. These properties also hold for another 4-truss community C_2. Notice that C_1 and C_2 are connected with, i.e., reachable from, each other, although they are not triangle connected with each other. If we had defined a k-truss community based on classical notion of connectivity, the union of the graphs C_1 and C_2 would be regarded a k-truss community. However, as the edges in C_1 cannot reach the edges in C_2 through adjacent triangles, C_1 and C_2 cannot merge as one large community. This is very reasonable, as there is no direct connection between the two vertex sets $\{p_1, p_2, p_3, p_4\}$ and $\{r_1, r_2, r_3\}$. Finally, we can see that vertices q and t participate in both communities C_1 and C_2.*

Problem Definition Given a graph $G(V, E)$, a query vertex $v_q \in V$, and an integer $k \geq 2$, find all k-truss communities containing v_q.

Why K-Truss Community?

To help appreciate the benefits of the k-truss community model, let us compare it with one of the most recent proposals for a community model, namely the α-adjacency-γ-quasi-k-clique model [54]. Compared with that model, the k-truss community model has three significant advantages: stronger guarantee on cohesive structure, fewer parameters, and lower computational cost. These nice properties, which are inherited from the k-truss subgraph [50], not only lead to the discovery of more cohesive and meaningful communities, but also enable the design of more efficient, scalable, and easier-to-use algorithms for community search. We elaborate these properties below.

Bounded Diameter in K-Truss Community. As shown in [50], the diameter of a k-truss community C with $|C|$ vertices is no larger than $\lfloor \frac{2|C|-2}{k} \rfloor$. This property guarantees that the shortest

distance between any two vertices in a community is bounded, which has been considered as an important feature of a good community in [61]. As an example, consider the 4-truss community C_1 in Figure 3.17b. The diameter of C_1 is 2, which matches the diameter upper bound $\lfloor \frac{2\times 6-2}{4} \rfloor = 2$.

(K-1)-Edge-Connected Graph. A graph is $(k-1)$-edge-connected if it remains connected whenever fewer than $k-1$ edges are removed [72]. A k-truss community is guaranteed to be $(k-1)$-edge-connected [50]. This property ensures a high connectivity of a community, which has been proposed as a criterion for a good community in [81]. In contrast, the γ-quasi-k-clique is not $(k-1)$-edge-connected whenever $\gamma < 1$. For example, the 0.8-quasi-7-clique in Figure 3.5 becomes disconnected when just one edge is removed.

Fewer Parameters. In the k-truss community model, we only need to specify the trussness value k, which controls the diameter, the triangle connectivity, and the edge support in a community. In contrast, the α-adjacency-γ-quasi-k-clique model requires three parameters, the adjacency parameter α, the density γ, and the clique size k. Arguably, one advantage of having more parameters may give more control over the properties of the community. On the other hand, it is much more difficult to set proper values for different parameters.

Polynomial Time Complexity. There exist polynomial time algorithms [50, 165] for computing k-truss subgraphs. By applying such algorithms, one can compute the k-truss subgraphs for all k. The precomputed results enable to design compact index structures and efficient algorithms for querying k-truss communities. In contrast, finding γ-quasi-k-cliques has been proven to be NP-hard [54], which imposes a severe computational bottleneck.

A Simple Index-Based Query Processing Algorithm

In the following, we discuss how to process a k-truss community query on a graph. We begin by describing a simple k-truss index and use it to develop a simple k-truss community search algorithm (Figure 3.18). We will subsequently analyze the limitations of this simple approach.

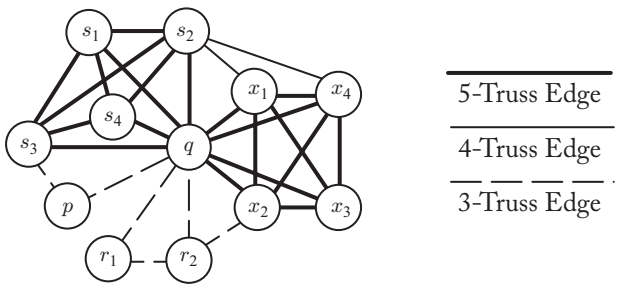


Figure 3.18: An example graph for k-truss community search.

Algorithm 3.11 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
4:         l ← l + 1; C_l ← ∅; L ← ∅;
5:         L.push((v_q, u)); visited ← visited ∪ {(v_q, u)};
6:         while L ≠ ∅
7:             (x, y) ← L.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
10:                    if (x, z) ∉ visited
11:                        L.push((x, z)); visited ← visited ∪ {(x, z)};
12:                    if (y, z) ∉ visited
13:                        L.push((y, z)); visited ← visited ∪ {(y, z)};
14: return {C_1, ..., C_l};
```

A Simple K-Truss Index Construction First, a truss decomposition algorithm [165] is applied to compute the trussness of all edges in G. Then, for each vertex $v \in V$, we sort its neighbors $u \in N(v)$ in descending order of the edge trussness $\tau(e(u, v))$. For each distinct trussness value $k \geq 2$, we mark the position of the first vertex u in the sorted adjacency list where $\tau(e(u, v)) = k$. This supports efficient retrieval of v's incident edges with a given trussness value. We also use a hashtable to maintain all the edges and their trussness values. This is the simple k-truss index.

Query Processing Algorithm 3.11 outlines the procedure for processing a k-truss community query based on the simple index. Given an integer k and a query vertex v_q, the algorithm checks every incident edge on v_q to search k-truss communities. If there exists an unvisited edge (v_q, u) with $\tau((v_q, u)) \geq k$, (v_q, u) is used as the seed edge to form a new community C_l. By definition, all the other edges in C_l should be reachable from (v_q, u) through adjacent triangles. So we push (v_q, u) into a queue L and perform a BFS traversal to search for other edges for expanding C_l, i.e., edges which have trussness no less than k and form triangles with edges already in C_l (lines 6–13). When L becomes empty, all edges in C_l have been found. Then the algorithm checks the next unvisited incident edge of v_q for forming a new community C_{l+1}. This process iterates until all incident edges of v_q have been processed. Finally, a set of k-truss communities containing v_q are returned.

The correctness of Algorithm 3.11 is apparent since the algorithm essentially computes k-truss communities by following the definition, that is, exploring triangle-connected edges with trussness no less than k in a BFS manner. We next show the complexity of the simple k-truss index construction and query processing by Algorithm 3.11.

Theorem 3.3.1 *The construction of the simple k-truss index takes $O(\sum_{(u,v)\in E} \min\{d(u), d(v)\})$ time and $O(m)$ space. The index size is $O(m)$. Algorithm 3.11 takes $O(d_{Amax}|Ans|)$ time to process one query, where $Ans = C_1 \cup \ldots \cup C_l$ is the union of the produced k-truss communities, $|Ans|$ is the number of edges in Ans and d_{Amax} is the maximum vertex degree in Ans.*

Proof. The truss decomposition algorithm (Algorithm 2.2) takes $O(\sum_{(u,v)\in E} \min\{d(u), d(v)\})$ time and $O(m)$ space for computing the trussness of all edges. Sorting the adjacency lists of all vertices in G can be done in $O(m)$ time and $O(m)$ space, using binsort, similarly to using the sorted degree array in [19, 37]. Building an edge hashtable costs $O(m)$ time and $O(m)$ space. Thus, the construction of the k-truss index takes $O(\sum_{(u,v)\in E} \min\{d(u), d(v)\})$ time and $O(m)$ space. The index size is $O(m)$.

In k-truss community search, for each edge (u, v) in the generated communities, Algorithm 3.11 accesses the common neighbors of u and v, i.e., $N(u) \cap N(v)$ (lines 7–9), whose size is bounded by d_{Amax}. Thus, the query time complexity is $O(d_{Amax}|Ans|)$. □

Example 3.3.2 *Suppose we want to query the 4-truss communities containing vertex q in the graph in Figure 3.18. Algorithm 3.11 first visits edge (q, s_1) with $\tau((q, s_1)) = 5 \geq 4$, and adds it into L. The algorithm pops (q, s_1) from L and inserts it into a new community C_1. Then the algorithm checks the common neighbors of q and s_1 and the edges between them. Consider a common neighbor s_2 as an example. As $\tau((q, s_2)) \geq 4$ and $\tau((s_1, s_2)) \geq 4$, both edges (q, s_2) and (s_1, s_2) are then inserted into C_1 and also pushed into L for further expansion. This BFS expansion process continues until L becomes empty and the 4-truss community C_1 is the subgraph induced by the vertex set $\{q, s_1, s_2, s_3, s_4, x_1, x_2, x_3, x_4\}$.*

TCP-Index-Based Query Processing Algorithms

In this section, we introduce a compact and elegant structure, called *Triangle Connectivity Preserving Index* (TCP-Index), and a highly efficient algorithm to process a k-truss community query. We first discuss the limitations of the simple k-truss index.

Limitations of Simple K-Truss Index Algorithm 3.11 has two drawbacks in its query processing mechanism of using the simple k-truss index. Specifically, in lines 8–13, for any edge (x, y) that has already been included in C_l, the algorithm needs to access adjacent edges (x, z) and (y, z) for each common neighbor z of x and y. The following two cases lead to unnecessary and excessive computational overhead.

1. **Unnecessary access of disqualified edges**: If $\tau((x, z)) < k$ or $\tau((y, z)) < k$, then $(x, z), (y, z)$ will not be included in C_l, thus accessing and checking such disqualified edges is clearly *wasteful* and should be avoided.

2. **Repeated access of qualified edges**: For each edge (u, v) in C_l, it is accessed at least $2(k - 2)$ times in the BFS traversal, which is a huge *overhead*, but avoidable. This is because $\tau((u, v)) \geq k$, (u, v) is contained in at least $(k - 2)$ triangles by definition. For each such

triangle denoted $\triangle_{uvw}$, (u, v) will be accessed twice when we do BFS expansion from the other two edges $(u, w), (v, w)$. It follows that the query time of Algorithm 3.11 is lower bounded by $\Omega(k|Ans|)$.

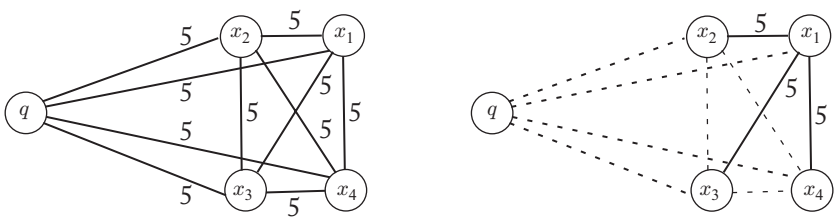
TCP-Index In view of these two drawbacks, Huang et al. [89] design a novel Triangle Connectivity Preserving Index, or TCP-Index for short, which avoids the computational issues of Algorithm 3.11 outlined above. Remarkably, the TCP-Index supports the k-truss community query in $O(|Ans|)$ time, which is essentially optimal. Meanwhile, the TCP-Index can be constructed in $O(\sum_{(u,v)\in E} \min\{d(u), d(v)\})$ time and stored in $O(m)$ space, which has exactly the same complexity as the simple k-truss index.

TCP-Index Construction

We first present some observations from the example in Figure 3.18.

Observation 3.3 Consider $\triangle_{pqs_3}$ in which the three edge trussness values are 5, 3, and 3. Then $\triangle_{pqs_3}$ can appear in a 3-truss community, but not in 4- or 5-truss communities. To generalize, a triangle $\triangle_{xyz}$ can appear only in k-truss communities where $k \leq \min\{\tau((x, y)), \tau((x, z)), \tau((y, z))\}$.

Observation 3.4 Consider the subgraph in Figure 3.19a, extracted from the graph in Figure 3.18. By definition, vertices x_1, x_2, x_3, x_4 all belong to the same 5-truss community containing q (see Figure 3.18), as each involved edge has trussness 5, and $\triangle_{qx_1x_2}$ and $\triangle_{qx_1x_3}$ are adjacent via edge (q, x_1). $\triangle_{qx_1x_2}$ and $\triangle_{qx_1x_3}$ are triangle connected. Similarly, $\triangle_{qx_1x_2}$, $\triangle_{qx_1x_3}$, $\triangle_{qx_1x_4}$, $\triangle_{qx_2x_3}$, $\triangle_{qx_2x_4}$ and $\triangle_{qx_3x_4}$ all are triangle connected. Thus, we can use a compact representation for vertex q as depicted in solid line in Figure 3.19b, which preserves the trussness and triangle connectivity information for community search. Note that there is no need to include edges (x_2, x_3), (x_2, x_4), and (x_3, x_4), as the tree-shaped structure clearly indicates that x_2, x_3, x_4 belong to the same 5-truss community as x_1 by triangle connectivity.



(a) Subgraph extracted from Figure 3.18 (b) Compact representation

Figure 3.19: Compact representation of a community with query vertex q.

Algorithm 3.12 TCP-Index Construction

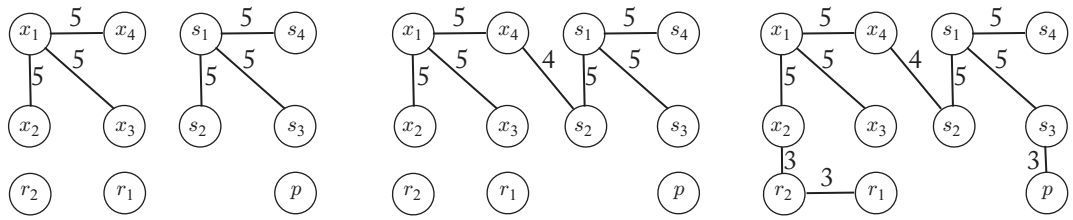
Input: $G = (V, E)$
Output: TCP-Index $\mathcal{T}_x$ for each $x \in V$

```
1: Perform truss decomposition for G;
2: for x ∈ V do
3:     G_x ← {(y,z)|y,z ∈ N(x),(y,z) ∈ E};
4:     for (y,z) ∈ E(G_x) do
5:         w(y,z) ← min{τ((x,y)),τ((x,z)),τ((y,z))};
6:     T_x ← N(x);
7:     k_max ← max{w(y,z)|(y,z) ∈ E(G_x)} ;
8:     for k ← k_max to 2 do
9:         S_k ← {(y,z)|(y,z) ∈ E(G_x),w(y,z) = k};
10:        for (y,z) ∈ S_k do
11:            if y and z are in different connected components in T_x
12:                add (y,z) with weight w(y,z) in T_x;
13: return {T_x|x ∈ V};
```

Observation 3.5 From Figure 3.18, we can see the two 5-truss communities $\{q, x_1, x_2, x_3, x_4\}, \{q, s_1, s_2, s_3, s_4\}$ involving vertex q are contained in the 4-truss community $\{q, x_1, x_2, x_3, x_4, s_1, s_2, s_3, s_4\}$, which is in turn contained in the 3-truss community, which is the whole graph.

Based on the above observations, Algorithm 3.12 outlines the procedure of constructing the TCP-Index. For each vertex $x \in V$, we build a graph G_x, where $V(G_x) = N(x)$, and $E(G_x) = \{(y, z)|(y, z) \in E(G), y, z \in N(x)\}$. For each edge $(y, z) \in E(G_x)$, we assign a weight $w(y, z) = \min\{\tau((x, y)), \tau((x, z)), \tau((y, z))\}$, which indicates that $\triangle_{xyz}$ can appear only in k-truss communities where $k \leq w(y, z)$, based on Observation 3.3. The TCP-Index for vertex x is a tree structure, denoted as $\mathcal{T}_x$, which is initialized to be the node set $N(x)$. Then in lines 8–12, for each k from the largest weight $k_{\max}$ to 2, we iteratively collect the set of edges $S_k \subseteq E(G_k)$ whose weight is k. For each $(y, z) \in S_k$, if y, z are still in different components of $\mathcal{T}_x$, we add the edge (y, z) with a weight $w(y, z)$ into $\mathcal{T}_x$. Essentially, $\mathcal{T}_x$ is the maximum spanning forest of G_x. The trees $\mathcal{T}_x$ for all $x \in V$ form the TCP-Index of graph G.

Example 3.3.3 *Figure 3.20 shows the* TCP-Index *for vertex q in the graph in Figure 3.18.* $\mathcal{T}_q$ *is initialized to be* $N(q)$. *Figure 3.20a shows the tree structure when we add edges whose weights are 5. According to Observation 3.4, when the edges* (x_1, x_2) *and* (x_1, x_3) *are added into* $\mathcal{T}_q$, *the edge* (x_2, x_3) *will not be added into* $\mathcal{T}_q$, *as* x_2, x_3 *are already connected in* $\mathcal{T}_q$ *and we know that* x_2, x_3 *belong to the same 5-truss community by triangle connectivity. This is essential to keep* $\mathcal{T}_q$ *as a compact forest. The complete* TCP-Index *for q is shown in Figure 3.20c. According to the community containment relationship in Observation 3.5, it is sufficient to use a single structure* $\mathcal{T}_q$ *for all trussness levels from* $k_{\max}$ *to 2.*



(a) 5-Truss level (b) 4-Truss level (c) 3-Truss level

Figure 3.20: TCP-Index construction of vertex q.

Theorem 3.3.2 *The* TCP-Index *of graph G can be constructed in $O(\sum_{(u,v)\in E} \min\{d(u), d(v)\})$ time and $O(m)$ space by Algorithm 3.12. The index size is $O(m)$.*

Proof. The first step costs $O(\sum_{(u,v)\in E} \min\{d(u), d(v)\})$ time. For a vertex $x \in V$, it takes $O(\sum_{y\in N(x)} \min\{d(x), d(y)\})$ time to list all triangles containing x to obtain G_x in line 3. The number of edges $|E(G_x)|$ is $O(\sum_{y\in N(x)} \min\{d(x), d(y)\})$, thus $\mathcal{T}_x$ can be computed in $O(\sum_{y\in N(x)} \min\{d(x), d(y)\})$ time by Kruskal's algorithm. For all vertices in V, it takes $O(\sum_{x\in V} \sum_{y\in N(x)} \min\{d(x), d(y)\})$ time in total to build the TCP-Index. Thus, the time complexity of Algorithm 3.12 is $O(\sum_{(u,v)\in E} \min\{d(u), d(v)\})$.

For a vertex $x \in V$, G_x, as a subgraph of G, takes $O(m)$ space, which can be released after obtaining $\mathcal{T}_x$. $\mathcal{T}_x$, as a spanning forest on the vertex set $N(x)$, takes $O(|N(x)|)$ space. Thus, the TCP-Index size for all vertices is $O(\sum_{x\in V} |N(x)|) = O(m)$. □

We remark that the *arboricity* of a graph is the minimum number of spanning forests needed to cover the edges of the graph. According to [47], $O(\sum_{(u,v)\in E} \min\{d(u), d(v)\}) \subseteq O(\rho m)$ where ρ is the arboricity of a graph G. $\rho \leq \min\{\lceil \sqrt{m} \rceil, d_{\max}\}$ holds for any graph. Thus, the TCP-Index construction takes $O(\sum_{(u,v)\in E} \min\{d(u), d(v)\}) \subseteq O(\rho m) \subseteq O(m^{1.5})$ time.

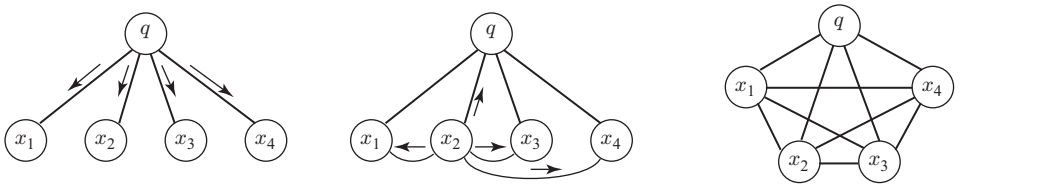
Query Processing Using TCP-Index

We first illustrate query processing through an example, before we formally present the algorithm. According to the design of the TCP-Index, if two vertices are connected through a series of edges with weight $\geq k$ in $\mathcal{T}_x$ for $x \in V$, these two vertices belong to the same k-truss community via a series of adjacent triangles. Consider $\mathcal{T}_q$ in Figure 3.20c. As x_2, x_3 are connected through two edges with weight 5, they belong to the same 5-truss community. Thus, we first define the *k-level connected vertex set* on a tree $\mathcal{T}_x$ to find all such vertices that belong to a k-truss community.

Definition 3.3.5 (K-Level Connected Vertex Set) *For $x \in V$ and $y \in N(x)$, we use $V_k(x, y)$ to denote the set of vertices which are connected with y through edges of weight $\geq k$ in $\mathcal{T}_x$. We adopt the convention that y also belongs to this set, i.e., $y \in V_k(x, y)$.*

Example 3.3.4 *If we want to query 5-truss communities containing a query vertex q, we first visit an incident edge on q, say (q, x_1), where $\tau((q, x_1)) = 5$. From $\mathcal{T}_q$ we retrieve the vertex set $V_5(q, x_1) = \{x_1, x_2, x_3, x_4\}$ as they are connected through edges with weight 5. According to Observation 3.4, these four vertices belong to the same 5-truss community with q. As $V_5(q, x_1) \subset N(q)$, we can construct part of the community as shown in Figure 3.21a.*

At this stage, we still miss the edges between the four vertices, for example, (x_2, x_3), (x_3, x_4), etc. This is because $\mathcal{T}_q$, which is a spanning forest, does not keep all the edges between the vertices. To fully recover all the edges in the 5-truss community, for each vertex $x_i \in V_5(q, x_1)$, we "reverse" the edge (q, x_i) to (x_i, q), then further expand the community in x_i's neighborhood. Take vertex x_2 as an example. We reverse (q, x_2) to (x_2, q) and then query x_2's index $\mathcal{T}_{x_2}$ to get the vertex set $V_5(x_2, q) = \{q, x_1, x_3, x_4\}$, as x_1, x_3, x_4 are connected with q in $\mathcal{T}_{x_2}$. Then we can obtain the edges between x_2 and every vertex in $V_5(x_2, q)$. After this, the community is shown in Figure 3.21b. Similarly, we perform the reverse operation for each vertex x_1, x_3, x_4 and get the complete 5-truss community in Figure 3.21c. We can observe that in this search process, each edge in a community is accessed exactly twice, for example, accessing (q, x_2) from $\mathcal{T}_q$ and (x_2, q) from $\mathcal{T}_{x_2}$.



(a) Expansion from q (b) Expansion from x_2 (c) Complete 5-truss community

Figure 3.21: 5-truss community query on q using TCP-Index.

Algorithm 3.13 outlines the procedure of query processing using the TCP-Index. Similar to Algorithm 3.11, Algorithm 3.13 computes the k-truss communities for a query vertex v_q by expanding from each incident edge on v_q in a BFS manner. If there exists an unvisited edge (v_q, u) with $\tau((v_q, u)) \geq k$, (v_q, u) is the seed edge to form a new community C_l (lines 2-4). Then the algorithm performs a BFS traversal using a queue L in lines 5-13. For an unvisited edge (x, y), it searches the vertex set $V_k(x, y)$ from $\mathcal{T}_x$. The procedure for computing $V_k(x, y)$ is listed in lines 15-16. For each $z \in V_k(x, y)$, the edge (x, z) is added into C_l. Then we perform the reverse operation, i.e., if (z, x) is not visited yet, it is pushed into L for z-centered community expansion using $\mathcal{T}_z$. Note that (z, x) and (x, z) are considered different here. When L becomes empty, all edges in C_l have been found. The process iterates until all incident edges of v_q have been processed. Finally, a set of k-truss communities containing v_q are returned.

We show the correctness of Algorithm 3.13 in the following.

Lemma 3.3.1 *Given a query vertex $x \in V$ and an integer k, Algorithm 3.13 correctly computes all k-truss communities containing x.*

Algorithm 3.13 Query Processing Using TCP-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
4:         l ← l + 1; C_l ← ∅; L ← ∅;
5:         L.push((v_q, u));
6:         while L ≠ ∅
7:             (x, y) ← L.pop();
8:             if (x, y) ∉ visited
9:                 compute V_k(x, y);
10:                for z ∈ V_k(x, y) do
11:                    visited ← visited ∪ {(x, z)}; C_l ← C_l ∪ {(x, z)};
12:                    if the reversed edge (z, x) ∉ visited
13:                        L.push((z, x));
14: return {C_1, ··· , C_l};
15: Procedure compute V_k(x, y)
16: return {z|z is connected with y in T_x through edges of weight ≥ k};
```

Proof. First, for an edge (y, z) in $\mathcal{T}_x$, by definition, $w(y, z) = \min\{\tau((x, y)), \tau((x, z)), \tau((y, z))\}$, so if $w(y, z) \geq k$, then $\triangle_{xyz}$ is included in a k-truss community of x.

Second, for two adjacent edges $(y, z_1), (y, z_2)$ in $\mathcal{T}_x$, we can conclude that $\triangle_{xyz_1}, \triangle_{xyz_2}$ are adjacent via edge (x, y).

Third, $V_k(x, y)$ contains the set of vertices which are connected with y through edges of weight $\geq k$ in $\mathcal{T}_x$. Based on the above two points, it leads to the discovery of all the triangles with weight $\geq k$ that can reach edge (x, y) in x's neighborhood. These connected triangles appear in the same k-truss community containing x.

Last, for an edge (x, z) where $z \in V_k(x, y)$, the same operation on its reverse edge (z, x) will further expand the k-truss community in z's neighborhood via $\mathcal{T}_z$. Thus, the k-truss community is expanded via adjacent triangles in a BFS manner.

The correctness of Algorithm 3.13 follows from the above points. □

Theorem 3.3.3 *The time complexity of Algorithm 3.13 is $O(|Ans|)$, where $Ans = C_1 \cup \ldots \cup C_l$ is the union of the produced k-truss communities and $|Ans|$ is the number of edges in Ans.*

Proof. Each edge (x, y) in the generated communities is accessed exactly twice: accessing (x, y) from $\mathcal{T}_x$ and (y, x) from $\mathcal{T}_y$. Thus, the time complexity of Algorithm 3.13 is $O(|Ans|)$. □

Complexity Comparison. By using the TCP-Index and the simple k-truss index, each edge in a k-truss community is accessed *exactly twice vs. at least* $2(k-2)$ *times*. In addition, the TCP-Index

successfully avoids the unnecessary access of disqualified edges whose trussness is less than k. These are the key reasons that explain the difference in the query time between Algorithms 3.13 and 3.11, i.e., $O(|Ans|)$ vs. $O(d_{Amax}|Ans|)$. It is worth noting that the TCP-Index construction has exactly the same time and space complexity as the simple k-truss index.

Case Study of k-truss and $(k-1,1)$-OCS Models on DBLP

We present a case study to compare the k-truss and $(k-1,1)$-OCS community models. A collaboration network is built from the DBLP data set[1] for this purpose. A vertex represents an author and an edge is added between two authors if they have co-authored three or more papers. The network contains 234,879 vertices and 541,814 edges.

We query the 5-truss community containing "Jiawei Han" which is shown in Figure 3.22. For comparison, we follow the case study in [54] which uses the $(k-1,1)$-OCS model to query "Jiawei Han" by setting $k=5, \alpha=4, \gamma=1$, which produces communities at a similar scale as shown in Figure 3.23. Note that we duplicate some authors who participate in more than one community in Figure 3.23, e.g., "Jian Pei", "Jian Pei_1" and "Jian Pei_2", for a better visualization effect. We have the following observations:

- The k-truss model generates five communities containing "Jiawei Han" (see Figure 3.22), among which the four smaller ones are also found by the $(k-1,1)$-OCS model and depicted using the same color in Figure 3.23.
- The largest 5-truss community depicted in blue in Figure 3.22, however, is decomposed into seven smaller communities by the $(k-1,1)$-OCS model in Figure 3.23. This phenomenon can be explained by the different mechanisms of the two community models. The $(k-1,1)$-OCS model tends to find the small, clique-based "*paper communities*," in which all the involved scholars appear in the same paper. For example, a paper community is formed by "Jiawei Han", "Philip S. Yu", "Chen Chen", "Xifeng Yan", and "Feida Zhu". In contrast, such small paper communities can be merged into a larger dense one by the condition of triangle connectivity in the k-truss model. For example, two small paper communities can be merged if they share a common edge as ("Jiawei Han", "Philip S. Yu") and form a 5-truss graph after being merged.
- A less restrictive community criterion can be realized by tuning α and γ in [54]. But in the experiment, if we set $\alpha < k-1$ or $\gamma < 1$, it cannot output all communities within the time limit of 60 seconds set in the executable code of [54], owing to the expensive quasi-clique enumeration.
- Finally, we observe a community containing "Guozhu Dong" and five other authors (depicted in purple) in Figure 3.23 is completely subsumed by another bigger community (depicted in black) in the same figure. Such duplicate output, which is not desired, may

[1] http://dblp.uni-trier.de/xml/

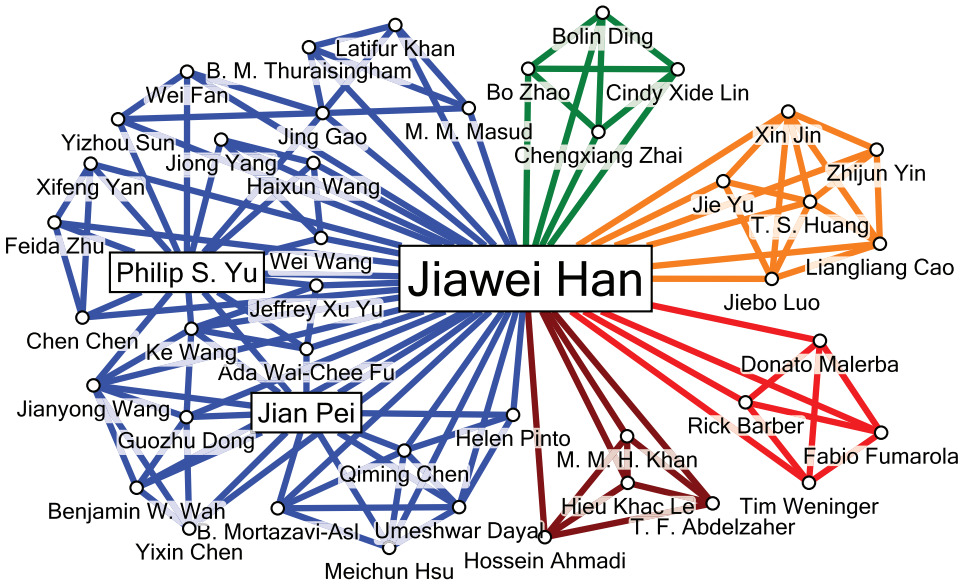


Figure 3.22: Five 5-truss communities containing Jiawei Han.

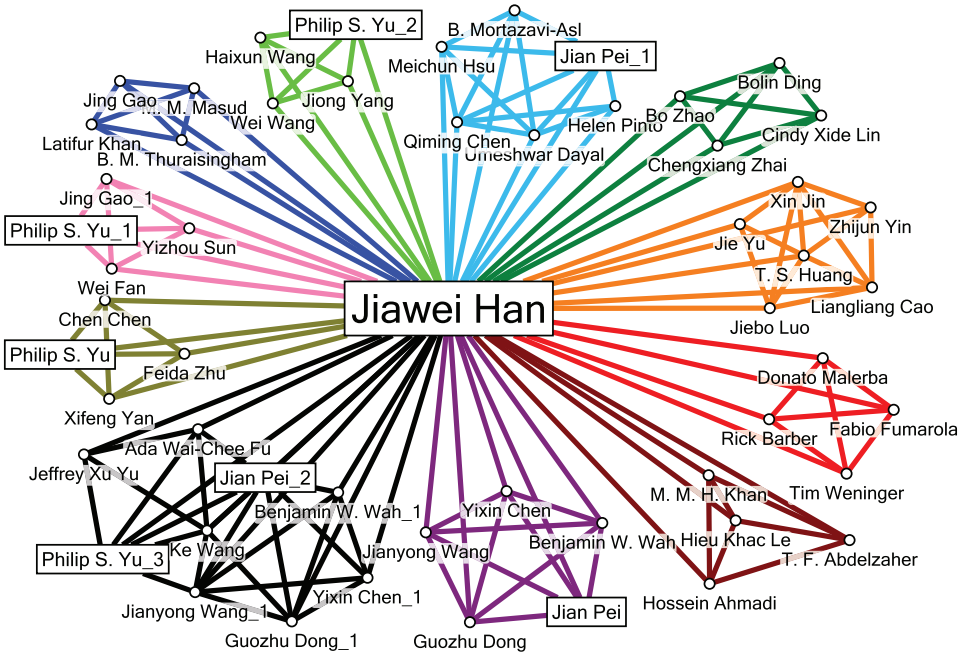


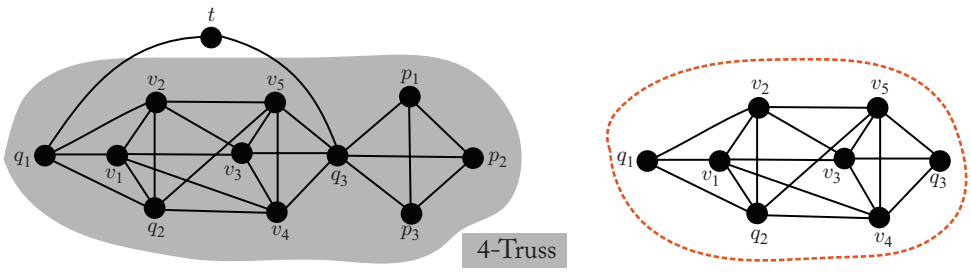
Figure 3.23: Eleven 4-adjacency-1.0-quasi-5-clique communities containing Jiawei Han.

be explained by the approximate heuristics for clique enumeration and expansion used in [54].

3.3.2 CLOSEST TRUSS COMMUNITY SEARCH

Motivation. In the k-truss community model discussed above, given one query node q and a parameter k, a k-truss community containing q is a maximal k-truss containing q, in which each edge is "triangle connected" with other edges. Triangle connectivity is strictly stronger than classical edge connectivity. The k-truss community model works well to find all overlapping communities containing a single query node q. It is natural to search for communities containing a set of query nodes in real applications, but the above community model, extended for multiple query nodes, has the following limitations. Due to the strict requirement of triangle connectivity constraint, the model may fail to discover any community for query nodes. For example, for the graph of Figure 3.24a and query nodes $Q = \{v_4, q_3, p_1\}$, the above k-truss community model cannot find a qualified community for any k, since the edges (v_4, q_3) and (q_3, p_1) are not triangle connected in any k-truss.

In this section, we present the problem of closest community search [96], i.e., given a set of query nodes, find a dense connected subgraph that contains the query nodes, in which nodes are close to each other. Based on graph diameter, we find the closest truss community containing query nodes with the smallest diameter.



(a) Graph G

(b) Closest Truss Community for $Q = \{q_1, q_2, q_3\}$

Figure 3.24: An example of closest truss communit.

Notions and Notations

For a pair of nodes $u, v \in G$, we denote by $\mathsf{dist}_G(u, v)$ the length of the shortest path between u and v in G, where $\mathsf{dist}_G(u, v) = +\infty$ if u and v are not connected. We make use of the notions of graph query distance and graph diameter.

Definition 3.3.6 (Query Distance) *Given a graph G and a set of query nodes $Q \subset V$, for each vertex $v \in G$, the vertex query distance of v is the maximum length of a shortest path from v to a query node $q \in Q$, i.e.,* $\mathsf{dist}_G(v, Q) = \max_{q \in Q} \mathsf{dist}_G(v, q)$. *For a subgraph $H \subseteq G$, the graph query distance of H is defined as* $\mathsf{dist}_G(H, Q) = \max_{u \in H} \mathsf{dist}_G(u, Q) = \max_{u \in H, q \in Q} \mathsf{dist}_G(u, q)$.

Definition 3.3.7 (Graph Diameter) *The diameter of a graph G is defined as the maximum length of a shortest path in G, i.e.,* $\mathsf{diam}(G) = \max_{u,v \in G} \{\mathsf{dist}_G(u, v)\}$.

For the graph G in Figure 3.24a and $Q = \{q_2, q_3\}$, the vertex query distance of v_2 is $\mathsf{dist}_G(v_2, Q) = \max_{q \in Q} \{\mathsf{dist}_G(v_2, q)\} = 2$, since $\mathsf{dist}_G(v_2, q_3) = 2$ and $\mathsf{dist}_G(v_2, q_2) = 1$. Let H be the subgraph of Figure 3.24a shaded in gray. Then the query distance of H is $\mathsf{dist}_G(H, Q) = 3$. The diameter of H is $\mathsf{diam}(H) = 4$.

Problem Formulation

On the basis of the definitions of k-truss and graph diameter, the *closest truss community* is defined as follows.

Definition 3.3.8 (Closest Truss Community) *Given a graph G and a set of query nodes Q, G' is a closest truss community (CTC), if G' satisfies the following two conditions.*

(1) Connected k-truss. *G' is a connected k-truss containing Q with the largest k, i.e., $Q \subseteq G' \subseteq G$ and $\forall e \in E(G')$, $sup(e) \geq k - 2$.*

(2) Smallest Diameter. *G' is a subgraph of smallest diameter satisfying condition (1). That is, $\nexists G'' \subseteq G$, such that $diam(G'') < diam(G')$, and G'' satisfies condition (1).*

Condition (1) requires that the closest community containing the query nodes Q be densely connected. In addition, Condition (2) makes sure that each node is as close as possible to every other node in the community, including the query nodes. We next illustrate the notion of CTC as well as the consequence of considering Conditions (1) and (2) in different orders.

Example 3.3.5 In Definition 3.3.8, we first consider the connected k-truss of G containing query nodes with the largest trussness, and then among such subgraphs, regard the one with the smallest diameter as the closest truss community. Consider the graph G in Figure 3.24a, and $Q = \{q_1, q_2, q_3\}$; the subgraph in the region shaded grey is a 4-truss containing Q, and is a subgraph with the largest trussness that contains Q, and has diameter 4. Notice that in Figure 3.24a, although the nodes p_1, p_2, p_3 belong to the 4-truss and are strongly connected with q_3, they are far away from the query node q_1. Figure 3.24b shows another 4-truss containing Q but not p_1, p_2, p_3, and its diameter is 3. It can be verified that this is the 4-truss containing the query nodes Q, that has the smallest diameter. Thus, by Condition (2) of Definition 3.3.8, the 4-truss graph in Figure 3.24a will not be regarded the closest truss community, whereas the one in Figure 3.24b is indeed the CTC.

Example 3.3.6 Suppose we apply the conditions in Definition 3.3.8 in the opposite order. That is, we first minimize the diameter among connected subgraphs of G containing Q and look for the k-truss subgraph with the largest k in that subgraph. First, we find that the cycle of $\{(q_1, t), (t, q_3), (q_3, v_4), (v_4, q_2), (q_2, q_1)\}$ is a connected subgraph containing Q with the smallest diameter 2. Then, we find that this cycle is also the k-truss subgraph with the largest k containing itself. However, it is only a 2-truss, which has a loosely connected structure compared to Figure 3.24b. It is left as a simple exercise to the reader to verify that other subgraphs with the same smallest diameter 2 that contain $Q = \{q_1, q_2, q_3\}$ do not admit k-trusses with $k > 2$. This justifies the choice of the order in which Conditions (1) and (2) should be applied.

The problem of CTC search is stated as follows.

Problem 3.3.1 (CTC-Problem**)** *Given a graph $G(V, E)$ and a set of query vertices $Q = \{v_1, \ldots, v_r\} \subseteq V$, find a closest truss community containing Q.*

Problem Analysis

Since the closest truss community model is based on the concept of k-truss, the communities capture good structural properties of k-truss, such as *k-edge-connected* and *hierarchical structure*. In addition, since CTC is required to have minimum diameter, it also has *bounded diameter*.

Small diameter, k-edge-connected, hierarchical structure. First, it has been shown that the diameter of a connected k-truss with n vertices is no more than $\lfloor \frac{2n-2}{k} \rfloor$ [50]. Moreover, a k-truss community is $(k-1)$-edge-connected [50], as it remains connected whenever fewer than $k-1$ edges are removed [72]. In addition, k-truss-based community has *hierarchical structure* that represents the essence of a community at different levels of granularity [89], that is, k-truss is always contained in the $(k-1)$-truss for any $k \geq 3$.

Largest k. There is a trivial upper bound on the maximum possible trussness of a connected k-truss containing the query nodes.

Lemma 3.3.2 *For a connected k-truss H satisfying definition of CTC for Q, we have $k \leq \min\{\tau(q_1), \ldots, \tau(q_r)\}$ holds.*

Proof. First, we have $Q \subseteq H$. For each node $q \in Q$, q cannot be contained in a k-truss in G, whenever $k > \tau(q)$. Thus, the fact that H is a k-truss subgraph containing Q implies that $k \leq \min\{\tau(q_1), \ldots, \tau(q_r)\}$. □

Lower and upper bounds on diameter. Since the distance function satisfies the triangle inequality, i.e., for all nodes u, v, w, $\mathsf{dist}_G(u, v) \leq \mathsf{dist}_G(u, w) + \mathsf{dist}_G(w, v)$, we can express the lower and upper bounds on the graph diameter in terms of the query distance as follows.

Lemma 3.3.3 *For a graph $G(V, E)$ and a set of nodes $Q \subseteq G$, we have $\mathsf{dist}_G(G, Q) \leq \mathsf{diam}(G) \leq 2\mathsf{dist}_G(G, Q)$.*

Proof. First, the diameter $\mathsf{diam}(G) = \max_{v,u \in G} \mathsf{dist}_G(v,u)$, which is clearly no less than $\mathsf{dist}_G(G, Q) = \max_{v \in G, q \in Q} \mathsf{dist}_G(v,q)$ for $Q \subseteq G$. Thus, $\mathsf{dist}_G(G, Q) \leq \mathsf{diam}(G)$. Second, suppose that the longest shortest path in G is between v and u. Then $\forall q \in Q$, we have $\mathsf{diam}(G) = \mathsf{dist}(v,u) \leq \mathsf{dist}(v,q) + \mathsf{dist}(q,u) \leq 2\mathsf{dist}_G(G, Q)$. The lemma follows. □

Hardness and Approximation

Hardness. In the following, we show the CTC-Problem is NP-hard. Thereto, we define the decision version of the CTC-Problem.

Problem 3.3.2 (CTCk-Problem) *Given a graph $G(V, E)$, a set of query nodes $Q = \{v_1, \ldots, v_r\} \subseteq V$ and parameters k and d, test whether G contains a connected k-truss subgraph with diameter at most d, that contains Q.*

Theorem 3.3.4 *The* CTCk-Problem *is* ***NP-hard****.*

Proof. We reduce the well-known NP-hard problem of Maximum Clique (decision version) to the CTCk-Problem. Given a graph $G(V, E)$ and a number k, the Maximum Clique Decision problem is to check whether G contains a clique of size k. Thus, we can construct an instance of the CTCk-Problem, consisting of graph G, $d = 1$, and $Q = \emptyset$.

We show that the instance of the Maximum Clique Decision problem is a YES-instance iff the corresponding instance of the CTCk-Problem is a YES-instance. Clearly, any clique with at least k nodes is a connected k-truss with diameter 1. On the other hand, given a solution H for the CTCk-Problem, H must contain at least k nodes since H is a k-truss, and $diam(H) = d = 1$, which implies H is a clique. □

The hardness of the CTC-Problem follows from this. The next natural question is whether the CTC-Problem can be approximated.

Approximation. For $\alpha \geq 1$, we say that an algorithm achieves an α-approximation to the CTC search problem if it outputs a connected k-truss subgraph $H \subseteq G$ such that $Q \subseteq H$, $\tau(H) = \tau(H^*)$ and $\mathsf{diam}(H) \leq \alpha \cdot \mathsf{diam}(H^*)$, where H^* is the optimal CTC. That is, H^* is a connected k-truss with the largest k s.t. $Q \subseteq H^*$, and $diam(H^*)$ is the minimum among all such CTCs containing Q. Notice that the trussness of the output subgraph H matches that of the optimal solution H^* and that the approximation is only w.r.t. the diameter: the diameter of H is required to be no more than $\alpha \cdot diam(H^*)$.

Non-Approximability. We next show that the CTC-Problem cannot be approximated within a factor better than 2. This result is established through a reduction, again from the Maximum Clique Decision problem to the problem of approximating the CTC-Problem, given k. In the next section, we describe a 2-approximation algorithm for the CTC-Problem, thus essentially matching this lower bound. Note that the CTC-Problem with given parameter k is essentially the CTCk-Problem.

Theorem 3.3.5 *Unless* P = NP, *for any* $\varepsilon > 0$, *the* CTC-Problem *with given parameter* k *cannot be approximated in polynomial time within a factor* $(2 - \varepsilon)$ *of the optimal.*

Proof Sketch: It can be shown that a $(2 - \varepsilon)$-approximation algorithm for the CTC-Problem with given parameter k can be used to distinguish between the YES and NO instances of the Maximum Clique Decision problem.

CTC Search Algorithm

In this section, we present a greedy algorithm called Basic for the CTC search problem. Then, we show that this algorithm achieves a 2-approximation to the optimal result.

Basic Algorithmic Framework. Here is an overview of our algorithm Basic. First, given a graph G and query nodes Q, we find a maximal connected k-truss, denoted G_0, containing Q and having the largest trussness. As G_0 may have a large diameter, we iteratively remove nodes far away from the query nodes, while maintaining the trussness of the remainder subgraph at k.

Algorithm. Algorithm 3.14 outlines the procedure of finding a closest truss community based on a greedy strategy. For query nodes Q, we first find a maximal connected k-truss G_0 that contains Q, such that $k = \tau(G_0)$ is the largest (line 1). Then, we set $l = 0$. For all $u \in G_l$ and $q \in Q$, we compute the shortest distance between u and q (line 4), and obtain the vertex query distance $\mathsf{dist}_{G_l}(u, Q)$. Among all vertices, we pick up a vertex u^* with the maximum $\mathsf{dist}_{G_l}(u^*, Q)$, which is also the graph query distance $\mathsf{dist}_{G_l}(G_l, Q)$ (lines 5–6). Next, we remove the vertex u^* and its incident edges from G_l, and delete any nodes and edges needed to restore the k-truss property of G_l (lines 7–8). We assign the updated graph as a new G_l. Then, we repeat the above steps until G_l does not have a connected subgraph containing Q (lines 3–9). Finally, we terminate by output graph R as the closest truss community, where R is any graph $G' \in \{G_0, \ldots, G_{l-1}\}$ with the smallest graph query distance $\mathsf{dist}_{G'}(G', Q)$ (line 10). Note that each intermediate graph $G' \in \{G_0, \ldots, G_{l-1}\}$ is a k-truss with the maximum trussness as required.

Example 3.3.7 *We apply Algorithm 3.14 on* G *in Figure 3.24 for* $Q = \{q_1, q_2, q_3\}$. *First, we obtain the 4-truss subgraph* G_0 *shaded in gray, using a procedure we will shortly explain. Then, we compute all shortest distances, and get the maximum vertex query distance as* $\mathsf{dist}_{G_0}(p_1, Q) = 4$, *and* $u^* = p_1$. *We delete node* p_1 *and its incident edges from* G_0; *we also delete* p_2 *and* p_3, *in order to restore the 4-truss property. The resulting subgraph is* G_1. *Any further deletion of a node in the next iteration of the while loop will induce a series of deletions in line 8, eventually making the graph disconnected or containing just a part of query nodes. As a result, the output graph* R, *shown in Figure 3.24b, is just* G_1. *Also* $\mathsf{dist}_R(R, Q) = 3$, *and* R *happens to be the exact CTC with diameter* 3, *which in this example is optimal.*

Approximation Analysis

Algorithm 3.14 can achieve a 2-approximation to the optimal solution, that is, the obtained connected k-truss community R satisfies $Q \subseteq R$, $\tau(R) = \tau(H^*)$ and $\mathsf{diam}(R) \leq 2\mathsf{diam}(H^*)$,

Algorithm 3.14 Basic (G, Q)

Input: A graph $G = (V, E)$, a set of query nodes $Q = \{q_1, \ldots, q_r\}$.
Output: A connected k-truss R with a small diameter.

1: Find a maximal connected k-truss containing Q with the largest k as G_0;
2: $l \leftarrow 0$;
3: **while** $\mathsf{connect}_{G_l}(Q) =$ **true do**
4: Compute $\mathsf{dist}_{G_l}(q, u)$, $\forall q \in Q$ and $\forall u \in G_l$;
5: $u^* \leftarrow \arg\max_{u \in G_l} \mathsf{dist}_{G_l}(u, Q)$;
6: $\mathsf{dist}_{G_l}(G_l, Q) \leftarrow \mathsf{dist}_{G_l}(u^*, Q)$;
7: Delete u^* and its incident edges from G_l;
8: Maintain k-truss property of G_l;
9: $G_{l+1} \leftarrow G_l$; $l \leftarrow l + 1$;
10: $R \leftarrow \arg\min_{G' \in \{G_0, \ldots, G_{l-1}\}} \mathsf{dist}_{G'}(G', Q)$;

for any optimal solution H^*. Since any graph in $\{G_0, \ldots, G_{l-1}\}$ is a connceted k-truss with the largest k containing Q by Algorithm 3.14, and $R \in \{G_0, \ldots, G_{l-1}\}$, we have $Q \subseteq R$ and $\tau(R) = \tau(H^*)$. In the following, we show that $\mathsf{diam}(R) \leq 2\mathsf{diam}(H^*)$. We start with a few key results. For graphs G_1, G_2, we say $G_1 \subseteq G_2$ to mean $V(G_1) \subseteq V(G_2)$ and $E(G_1) \subseteq E(G_2)$.

Lemma 3.3.4 *Given two graphs G_1 and G_2 with $G_1 \subseteq G_2$, for $u, v \in V(G_1)$, $\mathsf{dist}_{G_2}(u, v) \leq \mathsf{dist}_{G_1}(u, v)$ holds. Moreover, if $Q \subseteq V(G_1)$, then $\mathsf{dist}_{G_2}(G_1, Q) \leq \mathsf{dist}_{G_1}(G_1, Q)$ also holds.*

Proof. It trivially follows from the fact that G_2 preserves the paths between the nodes in G_1. □

Recall that in Algorithm 3.14, in each iteration i, a node u^* with the maximum $\mathsf{dist}(u^*, Q)$ is deleted from G_i, but $\mathsf{dist}_{G_i}(G_i, Q)$ is *not* monotonously nonincreasing during the process, hence $\mathsf{dist}_{G_{l-1}}(G_{l-1}, Q)$ is not necessarily the minimum. Note that in Algorithm 3.14, G_l is not the last feasible graph (i.e., connected k-truss containing Q), but G_{l-1} is. The observation is shown in the following lemma.

Lemma 3.3.5 *In Algorithm 3.14, it is possible that for some $0 \leq i < j < l$, we have $G_j \subset G_i$, and $\mathsf{dist}_{G_i}(G_i, Q) < \mathsf{dist}_{G_j}(G_j, Q)$ holds.*

Proof. It is easy to see that, because for a vertex $v \in G$, $\mathsf{dist}_G(v, Q)$ is non-decreasing monotone w.r.t. subgraphs of G. More precisely, for $v \in G_i \cap G_j$, $\mathsf{dist}_{G_i}(v, Q) \leq \mathsf{dist}_{G_j}(v, Q)$ holds. □

An important observation is that if an intermediate graph G_i obtained by Algorithm 3.14 contains an optimal solution H^*, i.e., $H^* \subset G_i$ and $\mathsf{dist}_{G_i}(G_i, Q) > \mathsf{dist}_{G_i}(H^*, Q)$, then the algorithm will not terminate at G_{i+1}.

Lemma 3.3.6 *In Algorithm 3.14, for any intermediate graph G_i, we have $H^* \subseteq G_i$ and $\mathsf{dist}_{G_i}(G_i, Q) > \mathsf{dist}_{G_i}(H^*, Q)$, then G_{i+1} is a connected k-truss containing Q and $H^* \subseteq G_{i+1}$.*

Proof. Suppose $H^* \subseteq G_i$ and $\text{dist}_{G_i}(G_i, Q) > \text{dist}_{G_i}(H^*, Q)$. Then there exists a node $u \in G_i \setminus H^*$ s.t. $\text{dist}_{G_i}(u, Q) = \text{dist}_{G_i}(G_i, Q) > \text{dist}_{G_i}(H^*, Q)$. Clearly, $u \notin Q$. In the next iteration, Algorithm 3.14 will delete u from G_i (Step 7), and perform Step 8. The graph resulting from restoring the k-truss property is G_{i+1}. Since H^* is a connected k-truss containing Q, the restoration step (line 8) must find a subgraph G_{i+1} s.t. $H^* \subseteq G_{i+1}$, and G_{i+1} is a connected k-truss containing Q. Thus, the algorithm will not terminate in iteration $(i + 1)$. □

The polynomial Algorithm 3.14 can find a connected k-truss community R having the minimum query distance to Q, which is optimal.

Lemma 3.3.7 *For any connected k-truss H with the highest k containing Q,* $\text{dist}_R(R, Q) \leq \text{dist}_H(H, Q)$.

Proof. The following cases may occur for G_{l-1}, which is the last feasible graph obtained by Algorithm 3.14.

Case (a): $H \subseteq G_{l-1}$. We have $\text{dist}_{G_{l-1}}(G_{l-1}, Q) \leq \text{dist}_{G_{l-1}}(H, Q)$; otherwise, if $\text{dist}_{G_{l-1}}(G_{l-1}, Q) > \text{dist}_{G_{l-1}}(H, Q)$, we can deduce from Lemma 3.3.6 that G_{l-1} is not the last feasible graph obtained by Algorithm 3.14, a contradiction. Thus, by Step 10 in Algorithm 3.14 and the fact that $\text{dist}_{G_{l-1}}(G_{l-1}, Q) \leq \text{dist}_{G_{l-1}}(H, Q)$, we have $\text{dist}_R(R, Q) \leq \text{dist}_{G_{l-1}}(G_{l-1}, Q) \leq \text{dist}_{G_{l-1}}(H, Q) \leq \text{dist}_H(H, Q)$.

Case (b): $H \not\subseteq G_{l-1}$. There exists a vertex $v \in H$ deleted from one of the subgraphs $\{G_0, \ldots, G_{l-2}\}$. Suppose the first deleted vertex $v^* \in H$ is in graph G_i, where $0 \leq i \leq l-2$, then v^* must be deleted in Step 7, but not in Step 8. This is because each vertex/edge of H satisfies the condition of k-truss, and will not be removed before any vertex is removed from G_i. Then, we have $\text{dist}_{G_i}(G_i, Q) = \text{dist}_{G_i}(v^*, Q) = \text{dist}_{G_i}(H, Q)$, and $\text{dist}_{G_i}(G_i, Q) \geq \text{dist}_R(R, Q)$ by Step 10. As a result, $\text{dist}_R(R, Q) \leq \text{dist}_{G_i}(H, Q) \leq \text{dist}_H(H, Q)$. □

Based on the preceding lemmas, we have the following.

Theorem 3.3.6 *Algorithm 3.14 provides a 2-approximation to the* CTC-Problem *as* $diam(R) \leq 2diam(H^*)$.

Proof. Since $\text{dist}_R(R, Q) \leq \text{dist}_{H^*}(H^*, Q)$ by Lemma 3.3.7, we get $diam(R) \leq 2\text{dist}_R(R, Q) \leq 2\text{dist}_{H^*}(H^*, Q) \leq 2diam(H^*)$ by Lemma 3.3.3. The theorem follows from this. □

Complexity Analysis

In the implementation of Algorithm 3.14, we do not need to keep all intermediate graphs, but just record the removal of vertices/edges at each iteration. Let G_0 be the maximal connected k-truss found in line 1 of Algorithm 3.14. Let $n' = |V(G_0)|$ and $m' = |E(G_0)|$, and let $d'_{\max}$ be the maximum degree of a vertex in G_0.

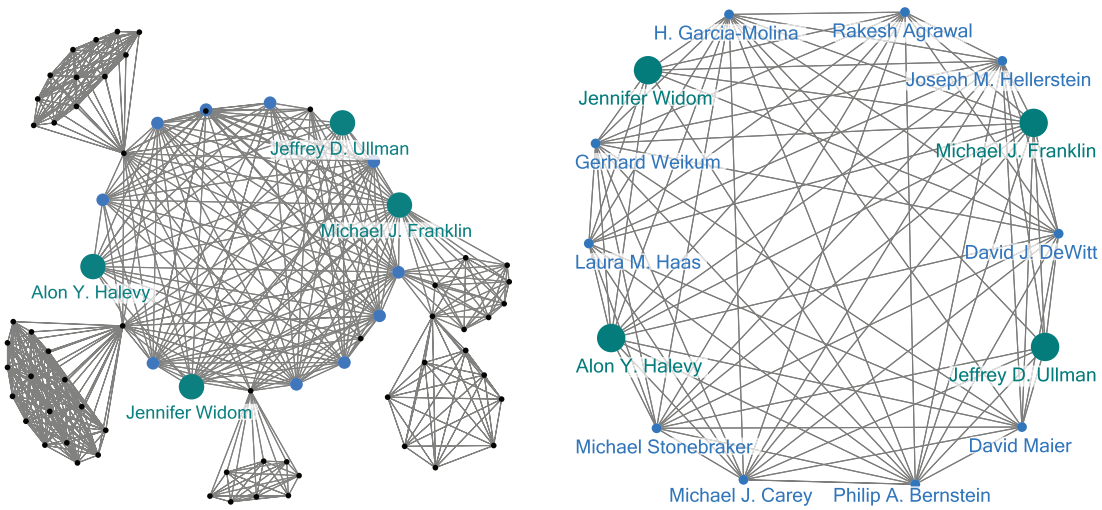
At each iteration i of Algorithm 3.14, we delete at least one node and its incident edges from G_i. Clearly, the number of removed edges is no less than $k - 1$, thus the total number of iterations is $t \leq \min\{n' - k, m'/(k-1)\}$, i.e., t is $O(\min\{n', m'/k\})$. We have the following.

Theorem 3.3.7 *Algorithm 3.14 takes* $O((|Q|t+\rho)m')$ *time and* $O(m')$ *space, where* $t \in O(\min\{n', m'/k\})$, *and* ρ *is the arboricity of graph* G_0. *Furthermore, we have* $\rho \leq \min\{d'_{\max}, \sqrt{m'}\}$.

Proof Sketch: First, listing all triangles of G_0 and creating a series of k-truss graphs $\{G_0, \ldots, G_{l-1}\}$ take $O(\rho m')$ time in all, where ρ is the arboricity of graph G_0. Second, the computation of shortest distances by a BFS traversal starting from each query node $q \in Q$ takes $O(t|Q|m')$ time for t iterations. Third, for the space consumption, we only record the sequence of removed edges from G_0 for attaching a corresponding label to graph G_i at each iteration i, which takes $O(m')$ space in all. We refer the reader to [97] for complete details of the proof.

Case Study

Figure 3.25b shows a closest truss community [96] detected on DBLP network using the query Q = {"Alon Y. Halevy," "Michael J. Franklin," "Jeffrey D. Ullman," "Jennifer Widom."} It has 14 authors, 81 edges, and an edge density of 0.89. Figure 3.25a shows a larger connected 9-truss containing the same query nodes, which includes the graph of Figure 3.25b as a subgraph. It is clear that this larger 9-truss includes several authors who are far away from and loosely connected with queried authors, and has an edge density of 0.18. This illustrates the superiority of closest truss community.



(a) 9-truss [50]

(b) Closesst truss community [96]

Figure 3.25: Community search on DBLP network using query Q ={"Alon Y. Halevy," "Michael J. Franklin," "Jeffrey D. Ullman," "Jennifer Widom."}

3.4 QUERY-BIASED DENSEST COMMUNITY MODEL

Motivation. Most community models find communities that contain a set of query nodes and also maximizes (or minimizes) a goodness metric. However, most models using goodness metrics tend to include irrelevant subgraphs in the detected communities. Such irrelevant subgraphs are referred as "free riders" in the literature. Wu et al. [180] introduce a query-biased node weighting scheme to reduce the free rider effect. The core idea of the proposed solutions is to give large node weights to such free riders far from query nodes, leading to a low density of a community involving these free riders. Finally, the communities with the largest density are returned as answers.

In this section, we discuss the community model based on the densest subgraphs developed by Wu et al. [180]. Given a graph G, the densest subgraph of G achieves the largest average degree among all possible subgraphs of G. Given a set of query nodes, the discovered communities should be densely connected in the local neighborhood of query nodes. In the following, we first use random walk-based proximity values to weight the nodes, with regard to the query nodes. The nodes farther away from the query nodes will have larger weights, which intuitively means that they are less important and thus will be levied more penalty in the calculation of density. After node weighting, we then introduce a new goodness metric of the query-biased density, and show that the query-biased densest subgraph is a target community in the neighborhood of the query node.

3.4.1 NOTIONS AND NOTATIONS

We first define the terms proximity, query-biased node weight, and query-biased density.

Proximity and Query-Biased Node Weight. To measure proximity, we use a variant of the degree normalized penalized hitting probability, which is referred to simply as the penalized hitting probability [180]. Let $w(u, v)$ be the edge weight between u and v, $w(u)$ be the degree of node u, and $w_{\max}$ be the maximum node degree. The transition probability from u to v is $\frac{w(u,v)}{w_{\max}}$, which is normalized by the maximum degree.[2] The penalized hitting probability penalizes the random walk for each additional step. The probability of hitting the query nodes for the first time is used as the proximity value. The penalized hitting probability can be defined as follows.

Definition 3.4.1 (Penalized Hitting Probability) *Given a graph $G(V, E)$ and a set of query nodes Q, the proximity value of $u \in V$ with regard to the query nodes Q is defined as*

$$r(u) = \begin{cases} 1, & \text{if } u \in Q; \\ c \sum_{v \in N_u} \frac{w(u,v)}{w_{\max}} \cdot r(v), & \text{if } u \in V - Q; \end{cases}$$

where c, $(0 < c < 1)$ is a decay factor.

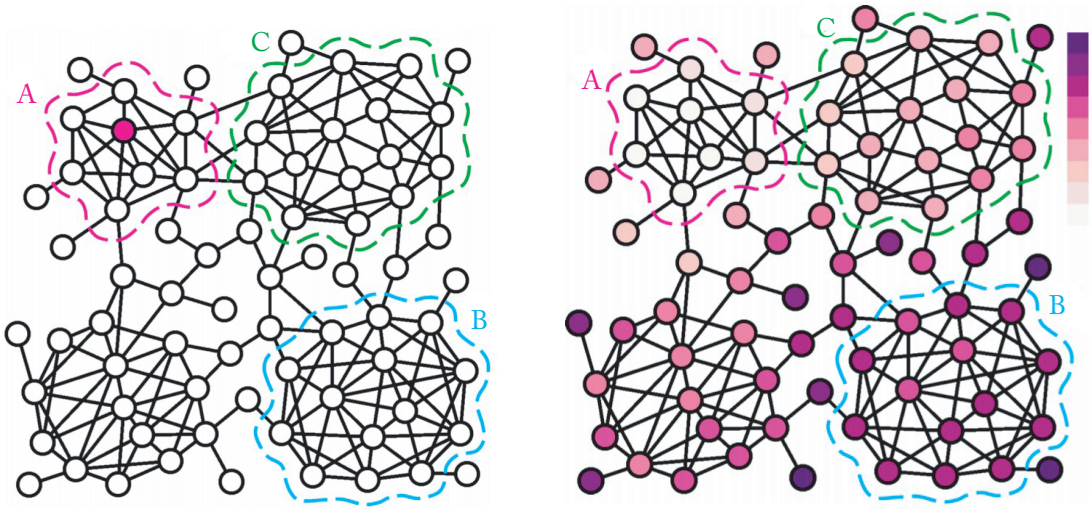
Note that the power iteration method [148] can be used to solve the above linear system in $O(\kappa m)$ time, where κ is the number of iterations.

[2]It follows that the weight of an edge can never exceed the maximum node degree in the graph.

Then, the query-biased node weight can be defined as follows.

Definition 3.4.2 (Query-Biased Node Weight) *The query-biased node weight $\pi(u)$ of node u is defined as the reciprocal of the penalized hitting probability $r(u)$, i.e., $\pi(u) = 1/r(u)$.*

From Definitions 3.4.1 and 3.4.2, we always have $0 \leq r(u) \leq 1$ and $\pi(u) \geq 1$. Consider the example of graph and query nodes in Figure 3.26a. The nodes in community A are densely connected to the query node through multiple short paths. Thus, the random walker will have a high probability of hitting the query node starting from any node in A. On the other hand, there are only a few long paths connecting the query node to the nodes not in A. Starting from these nodes, a random walker will have low probabilities to hit the query node, since the probabilities are penalized by the path lengths. Thus, the nodes in A will have higher proximity values than the nodes not in A. The distribution of the node weights, i.e., the reciprocal of the proximity values (i.e., the penalized hitting probabilities), is shown in Figure 3.26b, where a lighter color indicates a higher proximity value.



(a) Graph and query nodes

(b) Effect of node weighting

Figure 3.26: Examples of query-biased densest subgraph. The query node is the purple node in Figure 3.26a; There exist three communities A, B, and C. Effect of node weighting with $c = 0.9$ in Figure 3.26b. Darker color represents higher node weight; Subgraph A is the query-biased densest subgraph. Figures are borrowed from [180]. Used with Permission.

Query-Biased Density. Based on the query-biased node weights, the query-biased density is defined as follows.

Definition 3.4.3 (Query-Biased Density) *Given a graph $G(V, E)$ and a set of query nodes Q, the query-biased density of a subgraph G_S of G induced by a set of nodes S is defined as*

$$\rho(S) = \frac{|E(S)|}{\pi(S)},$$

where $|E(S)|$ is the number of edges in the induced subgraph G_S, and $\pi(S) = \sum_{u \in S} \pi(u)$ is the sum of the query-biased weights of nodes in S.

If the node weights $\pi(u) = 1$, the query-biased density degenerates to the classic density $\frac{e(S)}{|S|}$. In the following, we use the query-biased density and density interchangeably if there is no ambiguity. After node weighting, the densest subgraph is "shifted" to, i.e., is biased toward, the neighborhood of the query nodes. For example, in the graph shown in Figure 3.26a, before node weighting, the densest subgraph is B. Figure 3.26b shows the node weights after applying the node weighting scheme. A darker (lighter) color represents a larger node weight (proximity). After node weighting, subgraph A becomes the query-biased densest subgraph, which is as desired.

3.4.2 PROBLEM FORMULATION

To make sure the discovered communities will be densely connected and close to the neighborhood of the query nodes, we require the query-biased densest subgraph to always (1) contain the query nodes and (2) be connected. As a result, the problem of query-biased densest community search (QDC) can be formulated as follows.

Problem 3.4.1 (Query-Biased Densest Community Search (QDC)) *Given a graph $G(V, E)$ and a set of query nodes Q, find an induced subgraph G_S such that*

(1) $Q \subseteq S$;

(2) $\rho(S)$ *is maximized; and*

(3) G_S *is connected.*

3.4.3 ALGORITHMS

Wu et al. [180] propose an efficient algorithm to solve the QDC problem. Because the problem of QDC is NP-hard [180], it is challenging to develop efficient algorithms for find optimal communities. Wu et al. [180] instead relax the constraint of connectivity and define a variant QDC′ of QDC, without the connectivity requirement. The intuition of this relaxation is as follows. If there exists an optimal answer to the QDC′ problem and the discovered community is connected, then this community is also an optimal solution to the QDC problem.

Problem 3.4.2 (QDC′) *Given a graph $G(V, E)$ and a set of query nodes Q, find an induced subgraph G_S such that*

Algorithm 3.15 The algorithmic framework for the QDC problem

Input: A graph $G = (V, E)$, a set of query nodes Q, a decay factor c.
Output: A query-biased densest subgraph G_S containing Q.

1: Compute the node weights for every node by Definition 3.4.2;
2: Compute the optimal solution G_S of the QDC′ problem by Algorithm 3.16;
3: **if** G_S is connected **then**
4: **return** G_S;
5: **if** G_S contains a connected component G_T containing query nodes Q and at least one non-query node **then**
6: **return** G_T;
7: Apply the Maximum Adjacency Search in Algorithm 3.17 to find a heuristic solution G_S to QDC;
8: **return** G_S;

(1) $Q \subseteq S$ *and*

(2) $\rho(S)$ *is maximized.*

Algorithmic Framework. The overall algorithm for the QDC problem is outlined in Algorithm 3.15. It first computes the optimal solution G_S to QDC′ using Algorithm 3.16 (line 2). If G_S is connected, G_S is also the optimal solution to QDC (lines 3–4) and is returned as the output. However, if G_S is disconnected but there exists a connected subgraph G_T of G_S such that G_T contains all query nodes of Q and at least one non-query node, this connected subgraph G_T is returned as an approximate solution to QDC (lines 5–6). Otherwise, we apply another heuristic algorithm to find a solution G_S to QDC using Algorithm 3.17 (lines 7–8). In the following, we present the details of Algorithms 3.16 and 3.17.

Algorithm for the QDC′ problem. Wu et al. [180] develop an exact polynomial time algorithm for the QDC′ problem, outlined in Algorithm 3.16. This algorithm uses a new graph operation called *subgraph contraction*, defined as follows.

Definition 3.4.4 (Subgraph Contraction) *Given a graph $G(V, E)$ and a set of nodes Q, the operation of contracting a subgraph G_Q of G into a supernode q results in a new graph $G'(V', E')$, where the node set $V' = (V - Q) \cup \{q\}$ and the weight of supernode q as $r(q) = \sum_{v \in Q} r(v)$. The edge set E' is constructed as follows.*

(1) Keep edge (u, v) and its weight $w(u, v)$ if $(u, v) \in E$ and $u \notin Q, v \notin Q$.

(2) Add an edge (u, q) with weight $w(u, q) = \sum_{v \in Q} w(u, v)$, if $u \notin Q$.

(3) Add a self-loop edge (q, q) with weight $w(q, q) = \sum_{v,u \in Q} w(u, v)$, if $w(q, q) > 0$.

The operation of subgraph contraction preserves the density of graph G and G'. That is, for any $S \subset V - Q$, subgraphs $G_{Q \cup S}$ and $G'_{\{q\} \cup S}$ have the same density, so do subgraphs G_S and G'_S. Note that $G_{Q \cup S}$ is the induced subgraph of G by vertex set $S \cup Q$. $G'_{q \cup S}$ is the induced

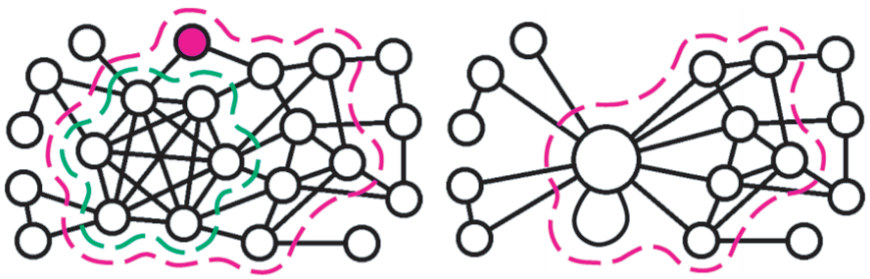
Algorithm 3.16 The algorithm for the QDC′ problem

Input: A graph $G = (V, E)$, a set of query nodes Q, node weights π.
Output: A query-biased densest subgraph G_S containing Q.

```
1: i ← 0; Q_0 ← Q; S ← ∅;
2: G_0 ← contract G_{Q_0} into a supernode q_0;
3: while true do
4:     Compute the densest subgraph G_{i_S} using the parametric maximum flow algorithm [149];
5:     if q_i ∈ S then
6:         S* ← S − {q_i} ∪ Q_i;
7:         break;
8:     Q_{i+1} ← S ∪ Q_i;
9:     G_{i+1} ← contract G_{Q_{i+1}} into q_{i+1};
10:    i ← i + 1;
11: return G_{S*};
```

subgraph of contraction graph G' by vertex set $q \cup S$, which also corresponds to the subgraph $G_{Q \cup S}$ in G.

Example 3.4.1 *Figure 3.27 shows an example of subgraph contraction [180]. In Figure 3.27a, each node and edge have the same unit weight in graph G. Note that the purple node in G is a query node. The densest subgraph of G is a 6-clique enclosed by the green curve, which has a density of 2.5. However, the densest subgraph does not include the query node, indicating it cannot be the answer of the* QDC *problem. We contract the densest subgraph along with the purple node into a supernode in the new graph G′ in Figure 3.27b. The densest subgraph of G′, enclosed by the purple curve, has a density of 2.38. This densest subgraph contains the supernode, indicating that it also contains the query node. Therefore, this densest subgraph is the optimal answer, which is also highlighted using the purple curve in Figure 3.27a.*



(a) Original graph G (b) Contracted graph G'

Figure 3.27: An example of subgraph contraction [180]. Used with Permission.

The procedure is outlined in Algorithm 3.16. Initially, the subgraph induced by the query nodes is contracted into a supernode (lines 1–2). In each iteration in lines 3–5, we find the

densest subgraph G_{iS} in G_i using the classical parametric maximum flow algorithm [149]. If the subgraph G_{iS} contains the supernode q_i, G_{S*} where the vertex set $S^* \leftarrow S - \{q_i\} \cup Q_i$, it is an optimal solution of the QDC′ problem and returned. Otherwise, it adds the set S into Q_i, which increases the density of the current subgraph G_{Q_i}. And then, it contracts $G_{S \cup Q_i}$ into a supernode and repeats this process until the densest subgraph G_{iS} contains the supernode q_i.

Complexity Analysis of Algorithm 3.16. Algorithm 3.16 runs in $O(\tau t)$ time, where τ is the number of iterations and t is the running time of solving the densest subgraph problem using maximum parametric flow [149]. Since at least one node is newly contracted into a supernode in each iteration, we have $\tau \leq n$ [180].

In the following, we describe the heuristic algorithm developed by Wu et al. [180] for the QDC problem. This is used when solving QDC′ does not give the desired solution.

Maximum Adjacency Search. Algorithm 3.17 presents the algorithm of maximum adjacent search, that is, to find a heuristic solution G_S to QDC. First, the algorithm uses Mehlhorn's algorithm [131] to compute the Steiner tree connecting all the given query vertices (line 1). As a result, the query vertices become connected together and this tree is used as the initial subgraph. When computing the Steiner tree, the edge weight is set to the reciprocal of the original edge weight. Next, the algorithm starts a local search process (lines 2–5). In each iteration, it finds a vertex u with the maximum adjacency value, i.e., $u \leftarrow \arg\max_{v \in V - S_i} w(\{v\}, S_i)/\pi(v)$. Finally, the intermediate subgraph with the maximum density during the local search process is returned as the query-biased densest subgraph. A parameter L is set to control the search space. When the vertex size of S_i is larger than L, the algorithm will terminate.

Complexity Analysis of Algorithm 3.17. The algorithm runs for at most L iterations. Let d_{avg} be the average degree of nodes. Then, for each iteration, it takes $O(i \cdot d_{avg})$ time to find a node with the maximum adjacency value (line 4 of Algorithm 3.17). As a result, the time complexity of the local search process is $O(\sum_{i=0}^{L}(i \cdot d_{avg})) \subseteq O(L^2 d_{avg})$. Finding the Steiner tree takes $O(m + n \log n)$ time [131].

Remarks. We note that the experimental results in [180] show that with more than 90% probability, Algorithm 3.15 gets the optimal solution of QDC by solving QDC′. With more than 5% probability, Algorithm 3.15 gets an approximate solution of QDC by solving QDC′. Therefore, only with less than 5% probability, Algorithm 3.15 needs to apply the heuristic algorithm in Algorithm 3.17 to find a solution of QDC. However, it needs to be borne in mind that this has no theoretical guarantee and that these findings are empirical.

3.5 SUMMARY

In this section, we summarize the various community models over simple graphs $G(V, E)$ that were discussed in this chapter. These community models are based on different dense subgraph definitions, such as clique [139], quasi-clique [54], densest subgraph [180], k-core [18, 55, 122, 157], and k-truss [89, 96]. In the following, we compare these models using metrics w.r.t. the

Algorithm 3.17 Maximum Adjacency Search Algorithm

Input: A graph $G = (V, E)$, a set of query nodes Q, node weights π, a parameter L.
Output: A query-biased densest subgraph G_S containing Q.

1: Compute the Steiner tree T connecting Q using the Mehlhorn's algorithm [131];
2: $S_0 \leftarrow$ the vertex set $V(T)$; $i \leftarrow 0$;
3: **while** $|S_i| \leq L$ **do**
4: $\quad u \leftarrow \arg\max_{v \in V - S_i} w(\{v\}, S_i)/\pi(v)$;
5: $\quad S_{i+1} \leftarrow S_i \cup \{u\}$;
6: $\quad i \leftarrow i + 1$;
7: $x \leftarrow \arg\max_i \rho(S_i)$; $S^* \leftarrow S_i$;
8: **return** G_{S^*};

following aspects: (i) consideration of query vertices, (ii) cohesive structure, (iii) index structure, (iv) query processing efficiency, and (v) quality of approximation. Table 3.2 shows a comparison of representative works on densely connected community search.

Query Vertices. Cui et al. [54] study the problem of online search of overlapping communities given a single query vertex, and design the α-adjacency γ-quasi-k-clique model. Huang et

Table 3.2: A comparison of representative works on cohesive community search. Here, "-" means that there exists no index for community search. Heuristic algorithms have non-approximate answers.

Method	Query Nodes	Cohesive Subgraph	Index Structure	Query Processing Efficiency	Quality Approximation
(α, γ)-OCS [54]	Single	α-adjacency-γ-quasi-k-clique	-	NP-hard	Exact
Global-Core [157]	Multiple	k-core	-	$O(n + m)$	Exact
Constrained-Core [157]	Multiple	k-core	-	NP-hard	Non-approximate
Local-Core [55]	Single	k-core	-	$O(n + m)$	Exact
Minimum-Core [18]	Multiple	k-core	Shell-Index	NP-hard	Non-approximate
k-Influential [122]	Multiple	k-core	ICP-Index	$O(\|Ans\|)$	Exact
Triangle-Connected-Truss [89]	Single	k-truss	TCP-Index	$O(\|Ans\|)$	Exact
Closest-Truss [96]	Multiple	k-truss	Truss-Index	NP-hard	2-approximation
QDC [180]	Multiple	Densest-subgraph	-	NP-hard	Non-guarantee-approximate

al. [89] propose a k-truss community model based on triangle connectivity to find all overlapping communities of a given query vertex. They ignore the diameter of the resulting community. Cui et al. [55] find a k-core community for a query vertex using local search. In addition, the influential community model [122] finds top-r communities with the highest influence scores over the entire graph; no query vertices are considered. Extending any of the above models from one (or zero) query vertex to multiple query vertices raises interesting challenges. The works [157], [18], [96], and [180] support community search with multiple query vertices.

Cohesive Structure. (α, γ)-OCS [54] is developed based on the α-adjacency-γ-quasi-k-clique. There exist several different models based on k-core subgraph, e.g., [157], [55], [18], and [122]. Sozio et al. [157] propose a k-core-based community model, called the Cocktail Party model, with distance and size constraints. The Triangle-Connected-Truss community model [89] and Closest-Truss community model [96] are based on the connected k-truss. Conceptually, k-truss is a more cohesive definition than k-core, as k-truss is based on triangles, where each "friendship" is endorsed by multiple common "friends," whereas k-core simply considers the vertex degree [165]. Most recently, Wu et al. [180] study the query-biased densest connected subgraph (QDC) problem for avoiding subgraphs irrelevant to query vertices in the community found. While QDC [180] is also defined based on a connected graph containing the query vertices similarly to Closest-Truss, it optimizes a fundamentally different function called query-biased edge density, which is calculated as the overall edge weight averaged over the weight of vertices in a community.

Index Structure and Query Processing Efficiency. Several studies [18, 89, 96, 122] propose indexes to speed up the query processing for their community search models. Barbieri et al. [18] design Shell-Index to quickly find a maximal connected k-core, and then solve the NP-hard problem of finding the minimum-sized community containing query nodes. Similarly, [122] proposes ICP-Index to find k-Influential communities in the optimal time cost of $O(|Ans|)$, where $|Ans|$ is the size of the answer community. Both Shell-Index and ICP-Index are designed for core-based community models. On the other hand, Huang et al. [89] design an elegant tree structure of TCP-Index to find the truss-based communities in the optimal time complexity of $O(|Ans|)$. The simple index of Truss-Index is used in [96] for finding a maximal connected k-truss containing all query nodes with the largest number of k, aiming at speeding up the query processing of approximation algorithms, due to NP-hardness of the problem. Other community models are not equipped with indexes, including (α, γ)-OCS [54], Global-Core [157], Constrained-Core [157], Local-Core [55], and QDC [180]. Among these methods without using indexes, Global-Core [157] and Local-Core [55] can find the optimal solution in time complexity of $O(n + m)$, whereas the other methods consider the NP-hard problems without proposing polynomial-time algorithms achieving some approximation guarantee.

Quality of Approximation. The problems proposed in (α, γ)-OCS [54], Constrained-Core [157], Minimum-Core [18], and QDC [180] are NP-hard to compute, and do not admit

approximations without further assumptions. [180] gives an approximation solution to QDC by relaxing the problem. Unfortunately, as shown in [180], this could fail in real applications, for two reasons. First, the algorithm may find a solution consisting of several connected components with query vertices split between them. Second, the approximation factor can be large, which can deteriorate further with a larger number of query vertices. In contrast, for Closest-Truss, there is an efficient 2-approximation algorithm for finding the closest truss community containing any set of query vertices.