Fast Algorithms for Intimate-Core Group Search in Weighted Graphs

Longxu Sun[1(✉)], Xin Huang[1], Rong-Hua Li[2], and Jianliang Xu[1]

[1] Hong Kong Baptist University, Hong Kong, China
sunlongxu@life.hkbu.edu.hk,{xinhuang,xujl}@comp.hkbu.edu.hk
[2] Beijing Institute of Technology, Beijing, China
lironghuabit@126.com

Abstract. Community search that finds query-dependent communities has been studied on various kinds of graphs. As one instance of community search, intimate-core group search over a weighted graph is to find a connected k-core containing all query nodes with the smallest group weight. However, existing state-of-the-art methods start from the maximal k-core to refine an answer, which is practically inefficient for large networks. In this paper, we develop an efficient framework, called local exploration k-core search (LEKS), to find intimate-core groups in graphs. We propose a small-weighted spanning tree to connect query nodes, and then expand the tree level by level to a connected k-core, which is finally refined as an intimate-core group. We also design a protection mechanism for critical nodes to avoid the collapsed k-core. Extensive experiments on real-life networks validate the effectiveness and efficiency of our methods.

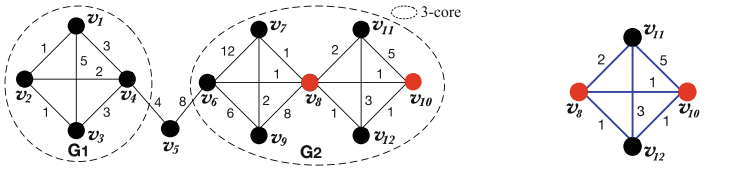
Keywords: Graph mining · Weighted graphs · K-core · Community search

1 Introduction

Graphs widely exist in social networks, biomolecular structures, traffic networks, world wide web, and so on. Weighted graphs have not only the simple topological structure but also edge weights. The edge weight is often used to indicate the strength of the relationship, such as interval in social communications, traffic flow in the transportation network, carbon flow in the food chain, and so on [18–20]. Weighted graphs provide information that better describes the organization and hierarchy of the network, which is helpful for community detection [19] and community search [10,11,13,26]. Community detection aims at finding all communities on the entire network, which has been studied a lot in the literature. Different from community detection, the task of community search finds only query-dependent communities, which has a wide application of disease infection control, tag recommendation, and social event organization [23,29]. Recently, several community search models have been proposed in different dense subgraphs of k-core [2,22] and k-truss [11,24].

R. Cheng et al. (Eds.): WISE 2019, LNCS 11881, pp. 728–744, 2019.
https://doi.org/10.1007/978-3-030-34223-4_46

As a notation of dense subgraph, k-core requires that every vertex has k neighbors in the k-core. For example, Fig. 1(a) shows a graph G. Subgraphs G_1 and G_2 are both connected 3-cores, in which each vertex has at least three neighbors. K-core has been popularly used in many community search models [1, 9,16,17,23,31]. Recently, Zheng et al. [29] proposed one problem of intimate-core group search in weighted graphs as follows.



(a) Graph G (b) Intimate-core group

Fig. 1. An example of intimate-core group search in graph G for $Q = \{v_8, v_{10}\}$ and $k = 3$.

Motivating Example. Consider a social network G in Fig. 1(a). Two individuals have a closer friendship if they have a shorter interval for communication, indicating a smaller weight of the relationship edge. The problem of intimate-core group search aims at finding a densely-connected k-core containing query nodes Q with the smallest group weight as an answer. For $Q = \{v_8, v_{10}\}$ and $k = 3$, the intimate-core group is shown in Fig. 1(b) with a minimum group weight of 13.

This paper studies the problem of intimate-core group search in weighted graphs. Given an input of query nodes in a graph and a number k, the problem is to find a connected k-core containing query nodes with the smallest weight. In the literature, existing solutions proposed in [29] find the maximal connected k-core and iteratively remove a node from this subgraph for intimate-core group refinement. However, this approach may take a large number of iterations, which is inefficient for big graphs with a large component of k-core. Therefore, we propose a solution of local exploration to find a small candidate k-core, which takes a few iterations to find answers. To further speed up the efficiency, we build a k-core index, which keeps the structural information of k-core for fast identification. Based on the k-core index, we develop a local exploration algorithm LEKS for intimate-core group search. Our algorithm LEKS first generates a tree to connect all query nodes, and then expands it to a connected subgraph of k-core. Finally, LEKS keeps refining candidate graphs into an intimate-core group with small weights. We propose several well-designed strategies for LEKS to ensure the fast-efficiency and high-quality of answer generations.

Contributions. Our main contributions of this paper are summarized as follows.

- We investigate and tackle the problem of intimate-core group search in weighted graphs, which has wide applications on real-world networks. The problem is NP-hard, which bring challenges to develop efficient algorithms.
- We develop an efficient local exploration framework of LEKS based on the k-core index for intimate-core group search. LEKS consists of three phases: tree generation, tree-to-graph expansion, and intimate-core refinement.
- In the phase of tree generation, we propose to find a seed tree to connect all query nodes, based on two generated strategies of *spanning tree* and *weighted path* respectively. Next, we develop the tree-to-graph expansion, which constructs a hierarchical structure by expanding a tree to a connected k-core subgraph level by level. Finally, we refine a candidate k-core to an intimate-core group with a small weight. During the phases of expansion and refinement, we design a protection mechanism for query nodes, which protects critical nodes to collapse the k-core.
- Our experimental evaluation demonstrates the effectiveness and efficiency of our LEKS algorithm on real-world weighted graphs. We show the superiority of our methods in finding intimate groups with smaller weights, against the state-of-the-art ICG-M method [29].

Roadmap. The rest of the paper is organized as follows. Section 2 reviews the previous work related to ours. Section 3 presents the basic concepts and formally defines our problem. Section 4 introduces our index-based local exploration approach LEKS. Section 5 presents the experimental evaluation. Finally, Sect. 6 concludes the paper.

2 Related Work

In the literature, numerous studies have been investigated community search based on various kinds of dense subgraphs, such as k-core [2,22], k-truss [11,24] and clique [25,26]. Community search has been also studied on many labeled graphs, including weighted graphs [7,29,30], influential graphs [4,16], and keyword-based graphs [8,9,12]. Table 1 compares different characteristics of existing community search studies and ours.

The problem of k-core minimization [1,6,17,31] aims to find a minimal connected k-core subgraph containing query nodes. The minimum wiener connector problem is finding a small connected subgraph to minimize the sum of all pairwise shortest-path distances between the discovered vertices [21]. Different from all the above studies, our work aims at finding an intimate-core group containing multiple query nodes in weighted graphs. We propose fast algorithms for intimate-core group search, which outperform the state-of-the-art method [29] in terms of quality and efficiency.

3 Preliminaries

In this section, we formally define the problem of intimate-core group search and revisit the existing intimate-core group search approaches.

Table 1. A comparison of existing community search studies and ours

Method	Dense subgraph model	Node type	Edge type	Local search	Index-based	Multiple query nodes	NP-hard
[25]	Clique	×	×	✓	✓	×	✓
[26]	Clique	×	×	×	✓	✓	✓
[14]	k-truss	×	×	✓	✓	✓	✓
[17,31]	k-core	×	×	×	×	×	✓
[6]	k-core	×	×	✓	×	×	✓
[23]	k-core	×	×	×	×	✓	✓
[1]	k-core	×	×	✓	✓	✓	✓
[12]	k-truss	Keyword	×	✓	✓	✓	✓
[9]	k-core	Keyword	×	✓	✓	×	✓
[16]	k-core	Influential	×	×	✓	×	×
[4]	k-core	Influential	×	✓	×	×	×
[30]	k-truss	×	Weighted	✓	✓	×	×
[29]	k-core	×	Weighted	×	×	✓	✓
Ours	k-core	×	Weighted	✓	✓	✓	✓

3.1 Problem Definition

Let $G(V, E, w)$ be a weighted and undirected graph where V is the set of nodes, E is the set of edge, and w is an edge weight function. Let $w(e)$ to indicate the weight of an edge $e \in E$. The number of nodes in G is defined as $n = |V|$. The number of edges in G is defined as $m = |E|$. We denote the set of neighbors of a node v by $N_G(v) = \{u \in V : (u, v) \in E\}$, and the degree of v by $deg_G(v) = |N_G(v)|$. For example, Fig. 1(a) shows a weighted graph G. Node v_5 has two neighbors as $N_G(v_5) = \{v_4, v_6\}$, thus the degree of v_5 is $deg_G(v_5) = 2$ in graph G. Edge (v_2, v_3) has a weight of $w(v_2, v_3) = 1$. Based on the definition of degree, we can define the k-core as follows.

Definition 1 (K-Core [2]). *Given a graph G, the k-core is the largest subgraph H of G such that every node v has degree at least k in H, i.e., $deg_H(v) \geq k$.*

For a given integer k, the k-core of graph G is denoted by $C_k(G)$, which is determinative and unique by the definition of largest subgraph constraint. For example, the 3-core of G in Fig. 1(a) has two components G_1 and G_2. Every node has at least 3 neighbors in G_1 and G_2 respectively. However, the nodes are disconnected between G_1 and G_2 in the 3-core $C_3(G)$. To incorporate connectivity into k-core, we define a connected k-core.

Definition 2 (Connected K-Core). *Given graph G and number k, a connected k-core H is a connected component of G such that every node v has degree at least k in H, i.e., $deg_H(v) \geq k$.*

Intuitively, all nodes are reachable in a connected k-core, i.e., there exist paths between any pair of nodes. G_1 and G_2 are two connected 3-cores in Fig. 1(a).

Definition 3 (Group Weight). *Given a subgraph $H \subseteq G$, the group weight of H, denoted by $w(H)$, is defined as the sum of all edge weights in H, i.e., $w(H) = \sum_{e \in E(H)} w(e)$.*

Example 1. For the subgraph $G_1 \subseteq G$ in Fig. 1(a), the group weight of G_1 is $w(G_1) = \sum_{e \in E(G_1)} w(e) = 1 + 3 + 5 + 2 + 1 + 3 = 15$.

On the basis of the definitions of connected k-core and group weight, we define the *intimate-core group* in a graph G as follows.

Definition 4 (Intimate-Core Group [29]). *Given a weighted graph $G = (V, E, w)$, a set of query nodes Q and a number k, the intimate-core group is a subgraph H of G if H satisfies following conditions:*

- ***Participation.*** *H contains all the query nodes Q, i.e., $Q \subseteq V_H$;*
- ***Connected K-Core.*** *H is a connected k-core with $deg_H(v) \geq k$;*
- ***Smallest Group Weight.*** *The group weight $w(H)$ is the smallest, that is, there exists no $H' \subseteq G$ achieving a group weight of $w(H') < w(H)$ such that H' also satisfies the above two conditions.*

Condition (1) of participation makes sure that the intimate-core group contains all query nodes. Moreover, Condition (2) of connected k-core requires that all group members are densely connected with at least k intimate neighbors. In addition, Condition (3) of minimized group weight ensures that the group has the smallest group weight, indicating the most intimate in any kinds of edge semantics. A small edge weight means a high intimacy among the group. Overall, intimate core groups have several significant advantages of small-sized group, offering personalized search for different queries, and close relationships with strong connections.

The problem of *intimate-core group search* studies in this paper is formulated in the following.

Problem Formulation: Given an undirected weighted graph $G(V, E, w)$, a number k, and a set of query nodes Q, the problem is to find the intimate-core group of Q.

Example 2. In Fig. 1(a), G is a weighted graph with 12 nodes and 20 edges. Each edge has a positive weight. Given two query nodes $Q = \{v_8, v_{10}\}$ and $k = 3$, the answer of intimate-core group for Q is the subgraph shown in Fig. 1(b). This is a connected 3-core, and also containing two query nodes $\{v_8, v_{10}\}$. Moreover, it has the minimum group weight among all connected 3-core subgraphs containing Q.

3.2 Existing Intimate-Core Group Search Algorithms

The problem of intimate-core group search has been studied in the literature [29]. Two heuristic algorithms, namely, ICG-S and ICG-M, are proposed to deal with this problem in an online manner. No optimal algorithms have been proposed yet because this problem has been proven to be NP-hard [29]. The NP-hardness is

shown by reducing the NP-complete clique decision problem to the intimate-core group search problem.

Existing solutions ICG-S and ICG-M both first identify a maximal connected k-core as a candidate, and then remove the node with the largest weight of its incident edges at each iteration [29]. The difference between ICG-S and ICG-M lies on the node removal. ICG-S removes one node at each iteration, while ICG-M removes a batch of nodes at each iteration. Although ICG-M can significantly reduce the total number of removal iterations required by ICG-S, it still takes a large number of iterations for large networks. The reason is that the initial candidate subgraph connecting all query nodes is the maximal connected k-core, which may be too large to shrink. This, however, is not always necessary. In particular, if there exists a small connected k-core surrounding query nodes, then a few numbers of iterations may be enough token for finding answers. This paper proposes a local exploration algorithm to find a smaller candidate subgraph. On the other hand, both ICG-S and ICG-M apply the core decomposition to identify the k-core from scratch, which is also costly expensive. To improve efficiency, we propose to construct an index offline and retrieve k-core for queries online.

4 Index-Based Local Exploration Algorithms

In this section, we first introduce a useful core index and the index construction algorithm. Then, we present the index-based intimate-core group search algorithms using local exploration.

4.1 K-Core Index

We start with a useful definition of coreness as follows.

Definition 5 (Coreness). *The coreness of a node $v \in V$, denoted by $\delta(v)$, is the largest number k such that there exists a connected k-core containing v.*

Obviously, for a node q with the coreness $\delta(q) = l$, there exists a connected k-core containing q where $1 \leq k \leq l$; meanwhile, there is no connected k-core containing q where $k > l$. The k-core index keeps the coreness of all nodes in G.

K-Core Index Construction. We apply the existing core decomposition [2] on graph G to construct the k-core index. The algorithm is outlined in Algorithm 1. The core decomposition is to compute the coreness of each node in graph G. Note that for the self-completeness of our techniques and reproducibility, the detailed algorithm of core decomposition is also presented (lines 1–7). First, the algorithm sort all nodes in G based on their degree in ascending order. Second, it finds the minimum degree in G as d. Based on the definition of k-core, it next computes the coreness of nodes with $deg_G(v) = d$ as d and removing these nodes and their incident edges from G. With the deletion of these nodes, the degree of neighbors of these nodes will decrease. For those nodes which have a new degree at most d, they will not be in (d+1)-core while they will get $\delta(v) = d$. It continues the

Algorithm 1. Core Index Construction

Input: A weighted graph $G = (V, E, w)$
Output: Coreness $\delta(v)$ for each $v \in V_G$

```
1: Sort all nodes in G in ascending order of their degree;
2: while G ≠ ∅
3:     Let d be the minimum degree in G;
4:     while there exists deg_G(v) ≤ d
5:         δ(v) ← d;
6:         Remove v and its incident edges from G;
7:         Re-order the remaining nodes in G in ascending order of their degree;
8: Store δ(v) in index for each v ∈ V_G;
```

removal of nodes until there is no node has $deg_G(v) \leq d$. Then, the algorithm back to line 2 and starts a new iteration to compute the coreness of remaining nodes. Finally, it stores the coreness of each vertex v in G as the k-core index.

4.2 Solution Overview

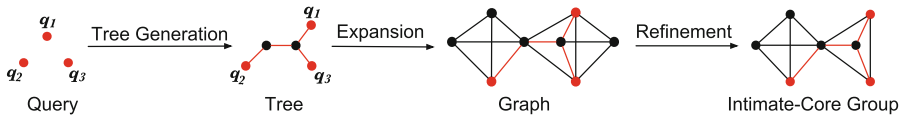


Fig. 2. LEKS framework for intimate-core group search

At a high level, our algorithm of local exploration based on k-core index for intimate-core group search (LEKS) consists of three phases:

1. *Tree Generation Phase*: This phase invokes the shortest path algorithm to find the distance between any pair of nodes, and then constructs a small-weighted tree by connecting all query nodes.
2. *Expansion Phase*: This phase expands a tree into a graph. It applies the idea of local exploration to add nodes and edges. Finally, it obtains a connected k-core containing all query nodes.
3. *Intimate-Core Refinement Phase*: This phase removes nodes with large weights, and maintains the candidate answer as a connected k-core. This refinement process stops until an intimate-core group is obtained.

Figure 2 shows the whole framework of our index-based local exploration algorithm. Note that we compute the k-core index offline and apply the above solution of online query processing for intimate-core group search. In addition, we consider $|Q| \geq 2$ for tree generation phase, and skip this phase if $|Q| = 1$. Algorithm 2 also depicts our algorithmic framework of LEKS.

Algorithm 2. LEKS Framework

Input: $G = (V, E, w)$, an integer k, a set of query vertices Q
Output: Intimate-core group H

1: Find a tree T_Q for query nodes Q using Algorithm 3 or Algorithm 4;
2: Expand the tree T_Q to a candidate graph G_Q in Algorithm 5;
3: Apply ICG-M [29] on graph G_Q;
4: Return a refined intimate-core group as answers;

4.3 Tree Generation

In this section, we present the phase of tree generation. Due to the large-scale size of k-core in practice, we propose local exploration methods to identify small-scale substructures as candidates from the k-core. The approaches produce a tree structure with small weights to connect all query nodes. We develop two algorithms, respectively based on the minimum spanning tree (MST) and minimum weighted path (MWP).

Tree Construction. The tree construction has three major steps. Specifically, the algorithm firstly generates all-pairs shortest paths for query nodes Q in the k-core C_k (lines 1–7). Given a path between nodes u and v, the path weight is the total weight of all edges along this path between u and v. It uses $\mathsf{spath}_{\mathcal{C}_k}(u, v)$ to represent the shortest path between nodes u and v in the k-core C_k. For any pair of query nodes q_i, $q_j \in Q$, our algorithm invokes the well-known Dijkstra's algorithm [5] to find the shortest path $\mathsf{spath}_{C_k}(q_i, q_j)$ in the k-core C_k.

Second, the algorithm constructs a weighted graph G_{pw} for connecting all query nodes (lines 3–8). Based on the obtained all-pairs shortest paths, it collects and merges all these paths together to construct a weighted graph G_{pw} correspondingly.

Third, the algorithm generates a small spanning tree for Q in the weighted graph G_{pw} (lines 9–22), since not all edges are needed to keep the query nodes connected in G_{pw}. This step finds a compact spanning tree to connect all query nodes Q, which removes no useful edges to reduce weights. Specifically, the algorithm starts from one of the query nodes and does expand based on Prim's minimum spanning tree algorithm [5]. The algorithm stops when all query nodes are connected into a component in G_{pw}. Against the maximal connected k-core, our compact spanning tree has three significant features: (1) Query-centric. The tree involves all query nodes of Q. (2) Compactly connected. The tree is a connected and compact structure; (3) Small-weighted. The generation of minimum spanning tree ensures a small weight of the discovered tree.

Example 3. Figure 3(a) shows a weighted graph G with 6 nodes and 8 edges with weights. Assume that $k = 2$, the whole graph is 2-core as C_2. A set of query nodes $Q = \{v_1, v_2, v_5\}$ are colored in red in Fig. 3(a). We first find the shortest path between every pair of query nodes in Q. All edges along with

Algorithm 3. Tree Construction

Input: $G = (V, E, w)$, an integer k, a set of query vertices Q, the k-core index
Output: Tree T_Q

```
1: Identify the maximal connected k-core of C_k containing query nodes Q;
2: Let G_pw be an empty graph;
3: for q_1, q_2 ∈ Q
4:     if there is no path between q_1 and q_2 in C_k then
5:         return ∅;
6:     else
7:         Compute the shortest path between q_1 and q_2 in C_k;
8:         Add the spath_{C_k}(q_1, q_2) between q_1 and q_2 into G_pw;
9: Tree: T_Q ← ∅;
10: Priority queue: L ← ∅;
11: for each node v in G_pw
12:     dist(v) ← ∞;
13: Q ← Q − {q_0}; dist (q_0) ← 0; L.push(q_0, dist(q_0));
14: while Q ≠ ∅ do
15:     Extract a node v and its edges with the smallest dist(v) from L;
16:     Insert node v and its edges into T_Q;
17:     if v ∈ Q then
18:         Q ← Q − {v};
19:     for u ∈ N_{G_pw}(v) do
20:         if dist(u) > w(u, v) then
21:             dist(u) ← w(u, v);
22:             Update (u, dist(u)) in L;
23: return T_Q;
```

(a) Find all pairs of shortest path (b) Spanning tree

Fig. 3. Tree construction for query nodes v_1, v_2, v_5.

these shortest path are colored in red in Fig. 3(a). For example, the shortest path between v_1 and v_2 is $\mathsf{spath}_{C_2}(v_1, v_2) = \{(v_1, v_3), (v_3, v_2)\}$. Similarly, $\mathsf{spath}_{C_2}(v_1, v_5) = \{(v_1, v_3), (v_3, v_4), (v_4, v_5)\}$, $\mathsf{spath}_{C_2}(v_2, v_5) = \{(v_2, v_5)\}$. All three paths are merged to construct a weighted graph G_{pw} in red in Fig. 3(a). A spanning tree of T_Q is shown in Fig. 3(b), which connects all query nodes $\{v_1, v_2, v_5\}$ with a small weight of 7.

Path-Based Construction. Algorithm 3 may take expensive computation for finding the shortest path between every pair of nodes that are far away from

Algorithm 4. Path-based Construction

Input: $G = (V, E, w)$, an integer k, a set of query vertices Q, the k-core index
Output: Tree T_Q

```
1: Identify the maximal connected k-core of C_k containing query nodes Q;
2: Let q_0 be the first query node of Q;
3: Q ← Q − {q_0};
4: while Q ≠ ∅ do
5:     if there is no path between q and q_0 in C_k then
6:         return ∅;
7:     else
8:         Compute the shortest path between q and q_0 in C_k;
9:         Add the spath_{C_k}(q, q_0) between q and q_0 into T_Q;
10:        q_0 ← q, Q ← Q − {q_0};
11: return T_Q;
```

each other. To improve efficiency, we develop a path-based approach to connect all query nodes directly. The path-based construction is outlined in Algorithm 4. The algorithm starts from one query node q_0, and searches the shortest path to the nearest query node in Q (lines 2–8). After that, it collects and merges the weighted path $\mathsf{spath}_{C_k}(q, q_0)$ into T_Q to construct the tree (line 9). Recursively, it starts from the new query node q as q_0 to find the next nearest query node q, until all query nodes in Q are found in such a way (line 10). The algorithm returns the tree connecting all query nodes.

Example 4. We apply Algorithm 4 on graph G in Fig. 3(a) with query $Q = \{v_1, v_2, v_5\}$ and $k = 2$. We start the shortest path search from v_1. The nearest query node to v_1 is v_5, we can find the shortest path $\mathsf{spath}_{C_2}(v_1, v_5) = \{(v_1, v_3), (v_3, v_4), (v_4, v_5)\}$. Next, we start from v_5 and find the shortest path $\mathsf{spath}_{C_2}(v_5, v_2) = \{(v_5, v_2)\}$. Finally, we merge the two paths $\mathsf{spath}_{C_2}(v_1, v_5)$ and $\mathsf{spath}_{C_2}(v_5, v_2)$ to construct the tree T_Q.

Complexity Analysis. We analyze the complexity of Algorithms 3 and 4. Assume that the k-core C_k has n_k nodes and m_k edges where $n_k \leq n$ and $m_k \leq m$.

For Algorithm 3, an intuitive implementation of all-pairs-shortest-paths needs to compute the shortest path for every pair nodes in Q, which takes $O(|Q|^2 m_k \log n_k)$ time. However, a fast implementation of single-source-shortest-path algorithm can compute the shortest path from one query node $q \in Q$ to all other nodes in Q, which takes $O(m_k \log n_k)$ time. Overall, the computation of all-pairs-shortest-paths can be done in $O(|Q| m_k \log n_k)$ time. In addition, the weighted graph G_{pw} is a subgraph of C_k, thus the size of G_{pw} is $O(n_k + m_k) \subseteq O(m_k)$. Identifying the spanning tree of G_{pw} takes $O(m_k \log n_k)$ time. Overall, Algorithm 3 takes $O(|Q| m_k \log n_k)$ time and $O(m_k)$ space.

For Algorithm 4, it applies $|Q|$ times of single-source-shortest-path to identify the nearest query node. Thus, Algorithm 4 also takes $O(|Q| m_k \log n_k)$ time and

Algorithm 5. Tree-to-Graph Expansion

Input: $G = (V, E, w)$, a set of query vertices Q, k-core index, T_Q
Output: Candidate subgraph G_Q

```
1: Identify the maximal connected k-core of C_k containing query nodes Q;
2: L_0 ← {v|v ∈ V_{T_Q}}; L' ← L_0 ;
3: i ← 0; G_Q ← ∅;
4: while G_Q = ∅ do
5:     for each v ∈ L_i do
6:         for each u ∈ N_{C_k}(v) and u ∉ L' ∪ L_{i+1} do
7:             L_{i+1} ← L_{i+1} ∪ {u};
8:     L' ← L' ∪ L_{i+1}; i ← i + 1;
9:     Let G_L be the induced subgraph of G by the node set L';
10:    Generate a connected k-core of G_L containing query nodes Q as G_Q;
11: return G_Q;
```

$O(m_k)$ space. In practice, Algorithm 4 runs faster than Algorithm 3 on large real-world graphs, which avoids the weighted tree construction and all-pairs-shortest-paths detection.

4.4 Tree-to-Graph Expansion

In this section, we introduce the phase of tree-to-graph expansion. This method expands the obtained tree from Algorithms 3 or 4 into a connected k-core candidate subgraph G_Q. It consists of two main steps. First, it adds nodes/edges to expand the tree into a graph layer by layer. Then, it prunes disqualified nodes/edges to maintain the remaining graph as a connected k-core. The whole procedure is shown in Algorithm 5.

Algorithm 5 first gets all nodes in T_Q and puts them into L_0 (line 2). Let L_i be the vertex set at the i-th depth of expansion tree, and L_0 be the initial set of vertices. It uses L' to represent the set of candidate vertices, which is the union of all L_i set. The iterative procedure can be divided into three steps (lines 4–10). First, for each vertex v in L_i, it adds their neighbors into L_{i+1} (lines 5–7). Next, it collects and merges $\{L_0, ..., L_{i+1}\}$ into L' and constructs a candidate graph G_L as the induced subgraph of G by the node set L' (lines 8–9). Finally, we apply the core decomposition algorithm on G_L to find the connected k-core subgraph containing all query nodes, denoted as G_Q. If there exists no such G_Q, Algorithm 5 explores the $(i+1)$-th depth of expansion tree and repeats the above procedure (lines 4–10). In the worst case, G_Q is exactly the maximum connected k-core subgraph containing Q. However, G_Q in practice is always much smaller than it. The time complexity for expansion is $O(\sum_{i=0}^{l_{max}} \sum_{v \in V(G_i)} \deg(v))$, where l_{max} is the iteration number of expansion in Algorithm 5.

(a) Expansion (b) Candidate subgraph construction

Fig. 4. Tree-to-graph expansion

Example 5. Figure 1(a) shows a weighted graph G with query $Q = \{v_8, v_{10}\}$ and $k = 3$. We first identify the maximal connected 3-core containing query nodes Q. Since there is only 2 query nodes, the spanning tree is same as the shortest path between them, such that $T_Q = \mathsf{spath}_{C_3}(v_8, v_{10})$. Next, we initialize L_0 as $L_0 = \{v_8, v_{10}\}$ and expand nodes in L_0 to their neighbors. The expansion procedure is shown in Fig. 4(a). We put all nodes in Fig. 4(a) into L' and construct a candidate subgraph G_L shown in Fig. 4(b). Since G_L is a 3-core connected subgraph containing query nodes, the expansion graph G_Q is G_L itself.

4.5 Intimate-Core Refinement

This phase refines the candidate connected k-core into an answer of the intimate-core group. We apply the existing approach ICG-M [29] by removing nodes to shrink the candidate graph obtained from Algorithm 5. This step takes $O(m' \log_{\varepsilon} n')$ time, where $\varepsilon > 0$ is a parameter of shrinking graph [29]. To avoid query nodes deleted by the removal processes of ICG-M, we develop a mechanism to protect important query nodes.

Protection Mechanism for Query Nodes. As pointed by [3,27,28], the k-core structure may collapse when critical nodes are removed. Thus, we precompute such critical nodes for query nodes in k-core and ensure that they are not deleted in any situations. We use an example to illustrate our ideas. For a query node q with an exact degree of k, it means that if any neighbor is deleted, there exists no feasible k-core containing q any more. Thus, q and all q's neighbors are needed to protect. For example, in Fig. 4(b), assume that $k = 3$, there exists $deg_G(v_{10}) = k$. The removal of each node in $N_G(v_{10})$ will cause core decomposition and the deletion of v_{10}. This protection mechanism for query nodes can also be used for k-core maintenance in the phrase of tree-to-graph expansion.

5 Experiments

In this section, we experimentally evaluate the performance of our proposed algorithms. All algorithms are implemented in Java and performed on a Linux server with Xeon E5-2630 (2.2 GHz) and 256 GB RAM.

Datasets. We use three real-world datasets in experiments. All datasets are publicly available from [15]. The edge weight represents the existence probability

of an edge. A smaller weight indicates a higher possibility of the edge to existing. The statistics of datasets are shown in Table 2. The maximum coreness $\delta_{max} = \max_{v \in V} \delta(v)$.

Table 2. Network statistics

Datasets	$\|V\|$	$\|E\|$	δ_{max}
wiki-vote	7,115	103,689	56
Flickr	24,125	300,836	225
DBLP	684,911	2,284,991	114

Algorithms. We compare 3 algorithms as follows.

- ICG-M: is the state-of-the-art approach for finding intimate-core group using bulk deletion [29].
- LEKS-tree: is our index-based search framework in Algorithm 2 using Algorithm 3 for tree generation.
- LEKS-path: is our index-based search framework in Algorithm 2 using Algorithm 4 for tree generation.

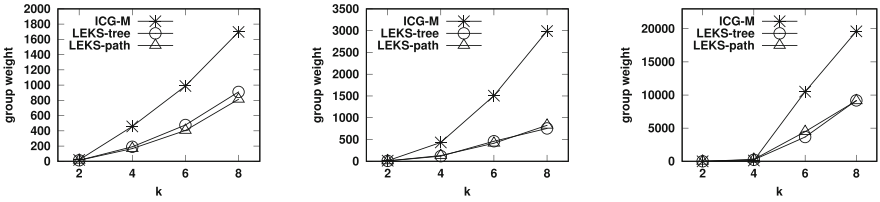
We evaluate all algorithms by comparing the running time and the intimate-core group weight. The less running time costs, the more efficient the algorithm is. Smaller the group weight of the answer, better effectiveness is.

Queries and Parameters. We evaluate all competitive approaches by varying parameters k and $|Q|$. The range of k is $\{2, 4, 6, 8\}$. The number of query nodes $|Q|$ falls in $\{1, 2, 3, 4, 5, 6, 7\}$. We randomly generate 100 sets of queries by different k and $|Q|$.

Exp-1: Varying k. Figure 5 shows the group weight of three algorithms by varying parameter k on all datasets. The results show that our local search methods LEKS-tree and LEKS-path can find intimate groups with lower group weights than ICG-M, for different k. The performance of LEKS-tree and LEKS-path are similar. Figure 6 shows that LEKS-path performs the best for most cases, and runs significantly faster than ICG-M. Interestingly, ICG-M can find answers quickly for $k = 4$, which achieves similar performance with LEKS methods.

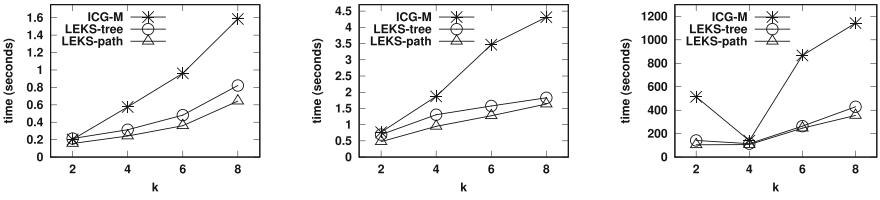
Exp-2: Varying $|Q|$. Figure 7 reports the group weight results of three algorithms for different queries by varying $|Q|$. With the increasing $|Q|$, LEKS-tree and LEKS-path methods can always find intimate groups with smaller weights than ICG-M. LEKS-tree and LEKS-path perform similarly. Figure 8 reports the results of running time. It shows that our methods are always faster than ICG-M.

Exp-3: Quality Evaluation of Candidate Intimate-Core Groups. This experiment evaluates the subgraphs of candidate intimate-core groups by all methods, in terms of vertex size and group weight. ICG-M takes the maximal



(a) wiki-vote (b) Flickr (c) DBLP

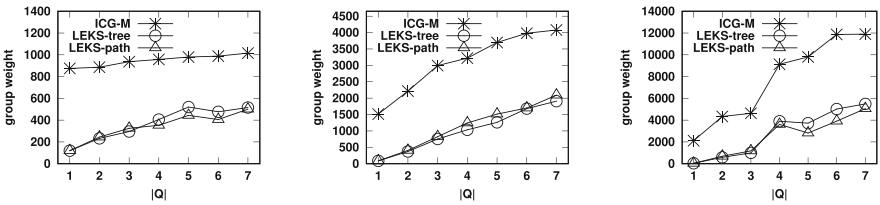
Fig. 5. Effectiveness evaluation by varying k



(a) wiki-vote (b) Flickr (c) DBLP

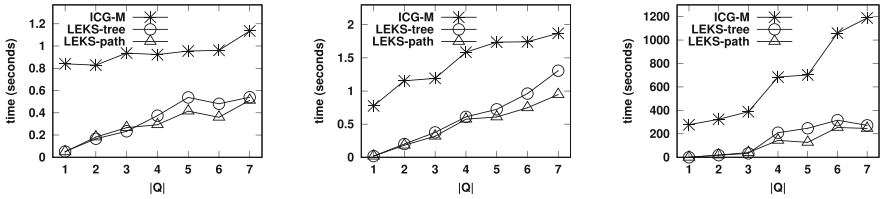
Fig. 6. Efficiency evaluation by varying k

connected k-core subgraph containing query nodes as an initial candidate, and iteratively shrinks it. LEKS-tree and LEKS-path both generate an initial candidate subgraph locally expanded from a tree, and then iteratively shrink the candidate by removing nodes. We use $k = 6$ and $|Q| = 5$. We report the results of the first 5 removal iterations and the initial candidate at the #iteration of 0. Figure 9(a) shows that the group weight of candidates by our methods is much smaller than ICG-M. Figure 9(b) reports the vertex size of all candidates at each iteration. The number of vertices in the candidate group by LEKS-tree and LEKS-path at the #iteration of 0, is even less than the vertex size of candidate group by ICG-M at the #iteration of 5.



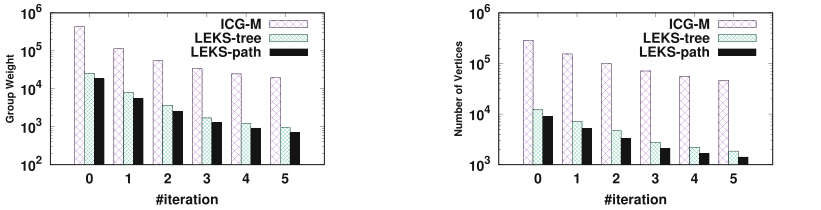
(a) wiki-vote (b) Flickr (c) DBLP

Fig. 7. Effectiveness evaluation by varying $|Q|$



(a) wiki-vote (b) Flickr (c) DBLP

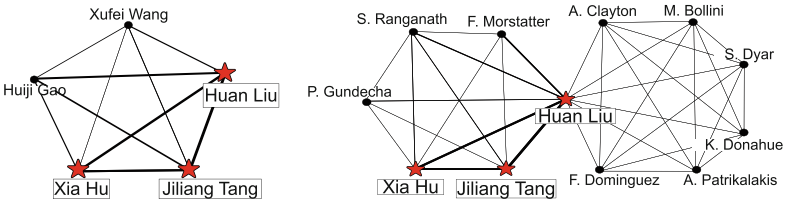
Fig. 8. Efficiency evaluation by varying $|Q|$



(a) Group weight varied by #iterations (b) Number of vertices varied by #iterations

Fig. 9. The size and weight of intimate-groups varied by #iterations

Exp-4: Case Study on the DBLP Network. We conduct a case study of intimate-core group search on the collaboration DBLP network [29]. Each node represents an author, and an edge is added between two authors if they have co-authored papers. The weight of an edge (u, v) is the reciprocal of the number of papers they have co-authored. The smaller weight of (u, v), the closer intimacy between authors u and v. We use the query $Q = \{$"Huan Liu", "Xia Hu", "Jiliang Tang"$\}$ and $k = 4$. We apply LEKS-path and ICG-M to find 4-core intimate groups for Q. The results of LEKS-path and ICG-M are shown in Fig. 10(a) and Fig. 10(b) respectively. The bolder lines of an edge represent a smaller weight, indicating closer intimate relationships. Our LEKS method discovers a compact 4-core with 5 nodes and 10 edges in Fig. 10(a), which has the group



(a) LEKS-path (b) ICG-M

Fig. 10. Case study of intimate-core group search on the DBLP network. Here, query $Q = \{$"Huan Liu", "Xia Hu", "Jiliang Tang"$\}$ and $k = 4$.

weight of 1.6, while ICG-M finds a subgraph with 12 nodes, which has a larger group weight of 16.7 in Fig. 10(b). We can see that nodes on the right side of Fig. 10(b) has no co-author connections with two query nodes "Xia Hu" and "Jiliang Tang" at all. This case study verifies that our LEKS-path can successfully find a better intimate-core group than ICG-M.

6 Conclusion

This paper presents a local exploration k-core search (LEKS) framework for efficient intimate-core group search. LEKS generates a spanning tree to connect query nodes in a compact structure, and locally expands it for intimate-core group refinement. Extensive experiments on real datasets show that our approach achieves a higher quality of answers using less running time, in comparison with the existing ICG-M method.

Acknowledgments. This work is supported by the NSFC Nos. 61702435, 61772346, U1809206, RGC Nos. 12200917, 12200817, and CRF C6030-18GF.

References

1. Barbieri, N., Bonchi, F., Galimberti, E., Gullo, F.: Efficient and effective community search. DMKD **29**(5), 1406–1433 (2015)
2. Batagelj, V., Zaversnik, M.: An O(m) algorithm for cores decomposition of networks. arXiv preprint arXiv:cs/0310049 (2003)
3. Bhawalkar, K., Kleinberg, J., Lewi, K., Roughgarden, T., Sharma, A.: Preventing unraveling in social networks: the anchored k-core problem. SIAM J. Discrete Math. **29**(3), 1452–1475 (2015)
4. Bi, F., Chang, L., Lin, X., Zhang, W.: An optimal and progressive approach to online search of top-k influential communities. PVLDB **11**(9), 1056–1068 (2018)
5. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to algorithms (2009)
6. Cui, W., Xiao, Y., Wang, H., Wang, W.: Local search of communities in large graphs. In: SIGMOD, pp. 991–1002 (2014)
7. Duan, D., Li, Y., Jin, Y., Lu, Z.: Community mining on dynamic weighted directed graphs. In: ACM International Workshop on Complex Networks Meet Information & Knowledge Management, pp. 11–18 (2009)
8. Fang, Y., Cheng, R., Chen, Y., Luo, S., Hu, J.: Effective and efficient attributed community search. VLDBJ **26**(6), 803–828 (2017)
9. Fang, Y., Cheng, R., Luo, S., Hu, J.: Effective community search for large attributed graphs. PVLDB **9**(12), 1233–1244 (2016)
10. Fang, Y., et al.: A survey of community search over big graphs. arXiv preprint arXiv:1904.12539 (2019)
11. Huang, X., Cheng, H., Qin, L., Tian, W., Yu, J.X.: Querying k-truss community in large and dynamic graphs. In: SIGMOD, pp. 1311–1322 (2014)
12. Huang, X., Lakshmanan, L.V.: Attribute-driven community search. PVLDB **10**(9), 949–960 (2017)

13. Huang, X., Lakshmanan, L.V., Xu, J.: Community Search over Big Graphs. Morgan & Claypool Publishers, San Rafael (2019)
14. Huang, X., Lakshmanan, L.V., Yu, J.X., Cheng, H.: Approximate closest community search in networks. PVLDB **9**(4), 276–287 (2015)
15. Huang, X., Lu, W., Lakshmanan, L.V.: Truss decomposition of probabilistic graphs: semantics and algorithms. In: SIGMOD, pp. 77–90 (2016)
16. Li, R.-H., Qin, L., Yu, J.X., Mao, R.: Influential community search in large networks. PVLDB **8**(5), 509–520 (2015)
17. Medya, S., Ma, T., Silva, A., Singh, A.: K-core minimization: a game theoretic approach. arXiv preprint arXiv:1901.02166 (2019)
18. Newman, M.E.: Scientific collaboration networks. II. Shortest paths, weighted networks, and centrality. Phys. Rev. E **64**(1), 016132 (2001)
19. Newman, M.E.: Analysis of weighted networks. Phys. Rev. E **70**(5), 056131 (2004)
20. Opsahl, T., Agneessens, F., Skvoretz, J.: Node centrality in weighted networks: generalizing degree and shortest paths. Soc. Netw. **32**(3), 245–251 (2010)
21. Ruchansky, N., Bonchi, F., García-Soriano, D., Gullo, F., Kourtellis, N.: The minimum wiener connector problem. In: SIGMOD, pp. 1587–1602 (2015)
22. Saríyüce, A.E., Gedik, B., Jacques-Silva, G., Wu, K.-L., Çatalyürek, Ü.V.: Streaming algorithms for k-core decomposition. PVLDB **6**(6), 433–444 (2013)
23. Sozio, M., Gionis, A.: The community-search problem and how to plan a successful cocktail party. In: KDD, pp. 939–948 (2010)
24. Wang, J., Cheng, J.: Truss decomposition in massive networks. PVLDB **5**(9), 812–823 (2012)
25. Yuan, L., Qin, L., Lin, X., Chang, L., Zhang, W.: Diversified top-k clique search. VLDBJ **25**(2), 171–196 (2016)
26. Yuan, L., Qin, L., Zhang, W., Chang, L., Yang, J.: Index-based densest clique percolation community search in networks. ICDE **30**(5), 922–935 (2017)
27. Zhang, F., Zhang, W., Zhang, Y., Qin, L., Lin, X.: OLAK: an efficient algorithm to prevent unraveling in social networks. PVLDB **10**(6), 649–660 (2017)
28. Zhang, F., Zhang, Y., Qin, L., Zhang, W., Lin, X.: Finding critical users for social network engagement: the collapsed k-core problem. In: AAAI (2017)
29. Zheng, D., Liu, J., Li, R.-H., Aslay, C., Chen, Y.-C., Huang, X.: Querying intimate-core groups in weighted graphs. In: IEEE International Conference on Semantic Computing, pp. 156–163 (2017)
30. Zheng, Z., Ye, F., Li, R.-H., Ling, G., Jin, T.: Finding weighted k-truss communities in large networks. Inf. Sci. **417**, 344–360 (2017)
31. Zhu, W., Chen, C., Wang, X., Lin, X.: K-core minimization: an edge manipulation approach. In: CIKM, pp. 1667–1670 (2018)