Efficient Influential Community Search over Dynamic Graphs

YOURAN SUN, The Chinese University of Hong Kong, Shenzhen, China
YINGLI ZHOU, The Chinese University of Hong Kong, Shenzhen, China
YIXIANG FANG*, The Chinese University of Hong Kong, Shenzhen, China
CHENG CHEN, ByteDance Inc, Singapore
YONGMIN HU, ByteDance Inc, China
YINGQIAN HU, ByteDance Inc, China

Influential community (IC) search has gained much attention with applications in many areas, such as event organization, recommendation systems, and biological analysis. The dynamic nature of graphs, with frequent insertions and deletions of vertices and edges, makes searching ICs over dynamic graphs computationally costly. To enable efficient IC search on large graphs, existing works often develop index-based approaches, which first compute all the ICs, a.k.a. IC decomposition, and then organize them into some index structures compactly. However, designed for static graphs, these index structures cannot be maintained efficiently for dynamic graphs, and little attention has been paid to the theoretical analysis of the index maintenance algorithms. To tackle the above issues, in this paper, we study the IC search problem on large dynamic graphs, and maintain the index structures efficiently by developing novel IC maintenance algorithms. We first theoretically show that all existing index maintenance algorithms, for the scenarios of both edge insertion and deletion, are relatively unbounded. We then propose a novel concept, called *IC decomposition order (ICD-order)*, based on which we further develop novel efficient IC maintenance algorithms for the scenarios of edge insertions and deletions, respectively. The comprehensive experiments on eight real datasets show that our algorithms are up to six and three orders of magnitude faster than state-of-the-art index maintenance algorithms under the edge insertion and deletion scenarios, respectively.

CCS Concepts: • **Human-centered computing** → **Social networks**; • **Mathematics of computing** → **Graph algorithms**; • **Theory of computation** → **Dynamic graph algorithms**.

Additional Key Words and Phrases: Community Search, Dynamic Graph

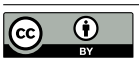
ACM Reference Format:
Youran Sun, Yingli Zhou, Yixiang Fang, Cheng Chen, Yongmin Hu, and Yingqian Hu. 2026. Efficient Influential Community Search over Dynamic Graphs. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 30 (February 2026), 28 pages. https://doi.org/10.1145/3786644

1 Introduction

In many real-world graphs (e.g., social networks and bibliographic networks), the vertices are often associated with importance values, denoting their influence values in the graph. For example, in the researcher collaboration network, the researchers' importance values can be reflected by their h-indexes, citation counts, or h5-indexes. Figure 1 gives a toy example, where the importance value

*Yixiang Fang is the corresponding author.

Authors' Contact Information: Youran Sun, The Chinese University of Hong Kong, Shenzhen, Shenzhen, China, youransun@link.cuhk.edu.cn; Yingli Zhou, The Chinese University of Hong Kong, Shenzhen, Shenzhen, China, yinglizhou@link.cuhk.edu.cn; Yixiang Fang, The Chinese University of Hong Kong, Shenzhen, Shenzhen, China, fangyixiang@cuhk.edu.cn; Cheng Chen, chencheng.sg@bytedance.com, ByteDance Inc, Singapore; Yongmin Hu, huyongmin@bytedance.com, ByteDance Inc, Beijing, China; Yingqian Hu, huyingqian@bytedance.com, ByteDance Inc, Hangzhou, China.



ACM 2836-6573/2026/2-ART30
https://doi.org/10.1145/3786644

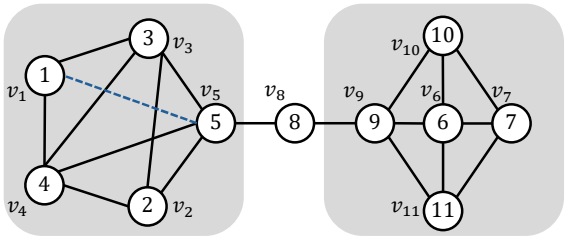


Fig. 1. An example graph with 11 vertices and 19 edges, and (v_1, v_5) is a new edge to be inserted.

of each vertex is an integer shown in the circle. Based on such graphs, a fundamental research topic is the *influential community (IC) search* [3, 12, 23, 39, 40, 42, 49, 56, 73, 81], which aims to find communities of vertices with high importance values from a graph, and has gained tremendous attention in the past decade. IC search has found many applications, such as extracting backbone structures from biology networks [7], event organization [3, 40, 42, 49, 81], and finding influencers in social networks [3, 40, 81].

Conceptually, an IC is a subgraph that is not only closely connected, but also has a high influence value. More specifically, an IC is a connected k-core, in which each vertex has at least k neighbors, and its influence value is defined as the minimum importance value of all vertices in this community [40]. In Figure 1, for example, given $k = 3$, the induced subgraph of $\{v_6, v_7, v_9, v_{10}, v_{11}\}$ is an IC whose influence value is 6. Given a graph G and two integers k and r, the IC search problem aims to identify the top-r ICs with the highest influence values from G. However, most existing IC search works focus on static graphs, and they cannot efficiently find ICs from large dynamic graphs, which are prevalent in many areas. In dynamic graphs, vertices and edges are frequently inserted and deleted. For instance, many new users have joined Facebook and established friendships from 2011 to 2024[1], resulting in a large dynamic network. *To tackle this issue, we aim to develop efficient algorithms for finding ICs in large dynamic graphs in this paper.*

To search ICs, an online algorithm is to first identify all the k-cores from the graph, then iteratively peels the vertices with minimum importance values from each k-core until the minimum degree constraint is not satisfied, and finally returns the r ICs with the highest influence values. However, this algorithm is inefficient for large graphs, so researchers have developed many index-based approaches in the literature [3, 40].

To build the indices, these approaches generally consist of three steps: 1) enumerate all the possible k values from 0 to its maximum value; 2) for each specific k, iteratively remove the vertex with the minimum importance value from the graph (marked as a keynode) until the minimum degree constraint is not satisfied; 3) consequently remove the vertices with degrees less than k (marked as *cvs* of the latest removed keynode) to meet the constraint again, and 4) organize all the ICs into some compact indices. The first two steps are also known as *IC decomposition (ICD)*, which computes all the possible ICs in the graph. For the third step, Li et al. [40] proposed the `ICP-Index`, which stores all the ICs sharing the same k value in a tree structure. Bi et al. [3] further proposed an improved index, `KC-Index`, based on the key observation that each IC consists of a keynode vertex (the vertex with minimum weight) and multiple *cvs* vertices. Specifically, `KC-Index` maintains a lightweight mapping from each keynode to its associated *cvs* vertices. Based on these indices, the top-r ICs can be efficiently derived, and both types of indices achieve comparable performance in IC search [3]. Consider the graph in Figure 1 with k=3, which has two connected 3-cores: $\{v_2, v_3, v_4, v_5\}$

[1]https://www.statista.com/statistics/346167/facebook-global-dau/

and $\{v_6, v_7, v_9, v_{10}, v_{11}\}$. We first obtain a keynode v_2, and after removing it, the vertices $\{v_3, v_4, v_5\}$ will be iteratively removed, so they are *cvs* vertices associated with v_2. Similarly, we can obtain another keynode v_6 and its corresponding *cvs* vertices $\{v_7, v_9, v_{10}, v_{11}\}$. By keeping the mapping between each keynode and its corresponding *cvs* vertices, we obtain the index in Figure 2.

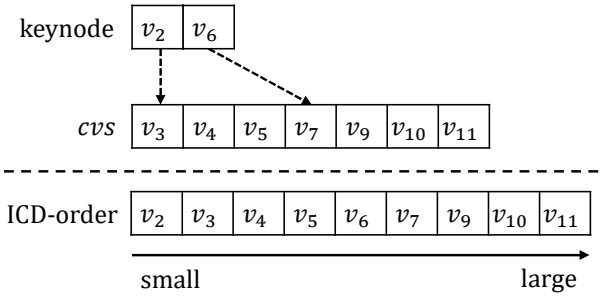


Fig. 2. The keynode and *cvs* vertices of the graph for $k = 3$.

Limitations of existing works. Although the above index-based approaches work well, they are inefficient for processing large dynamic graphs because maintaining the above index structures is not only costly but also non-trivial. To our best knowledge, [42] is the only work that has studied this problem. In [42], Li et al. proposed some algorithms to maintain the `ICP-Index` for dynamic graphs, and the core idea is to rebuild the `ICP-Index` when a new edge is inserted or an existing edge is deleted, with some pruning techniques. The key reason is that when a new edge is inserted or an existing edge is deleted, only a small number of ICs are changed, while the above algorithms almost rebuild the whole `ICP-Index`, which fails to accurately locate these unchanged parts, resulting in significant redundant computations.

Besides, none of the existing works has analyzed the boundedness of the `ICP-Index` maintenance algorithms, so it is not clear whether the change of index is bounded or not when the graph is changed. To fill this gap, we theoretically prove that they are relatively unbounded under both edge insertion and deletion scenarios. In other words, when an edge is inserted or deleted, they cannot bound their complexity by the size of the changed part of the `ICP-Index`, which may propagate across the entire `ICP-Index`, leading to a huge time cost of maintenance.

Our approaches. Compared to `ICP-Index`, `KC-Index` is more lightweight and tends to be easier for maintenance, since it only requires tracking changes in vertex types and their corresponding mappings during graph changes. In light of this, we focus on maintaining the `KC-Index` to support efficient IC search in large dynamic graphs. To achieve this, we first introduce a novel concept, called *IC decomposition order* or *ICD-order*, which represents the vertex removal sequence in the ICD process. For example, consider the graph in Figure 1 with $k = 3$, the ICD-order is $\{v_2, v_3, v_4, v_5, v_6, v_7, v_9, v_{10}, v_{11}\}$. Based on ICD-order, we prove that both keynodes and their associated *cvs* vertices can be directly extracted from the vertex sequence according to their positions in the ICD-order. In other words, maintaining the ICD-order under edge insertions and deletions can be considered as maintaining the keynodes and *cvs* vertices under the same dynamic scenarios.

To maintain the `KC-Index` for dynamic graphs, we need to maintain the ICD-order for all k values. More specifically, for each k value, we first identify the candidate set, which is defined as the union of all vertices whose positions may change in the ICD-order and those whose core numbers change, and then iteratively execute the following two steps until the candidate set is empty: 1) find the *navigation node*, which is the first vertex in the ICD-order whose type or position may change due to edge insertion or deletion, and append it to the candidate set for the subsequent

process if it changes; and 2) based on the navigation node and our carefully designed update rules, determine which vertices can be placed into their correct positions in the new ICD-order, and remove them from the candidate set. While the KC-Index maintenance algorithms for the scenario of edge insertion and deletion share the same core ideas, they differ in how the navigation node is identified and how the specific update rules are designed. Compared to existing methods, our order-based IC maintenance algorithms provide the first relatively bounded solution for the edge insertion scenario and achieve better time complexity for the edge deletion scenario.

We have performed extensive experimental evaluations on eight real-world dynamic graphs, and the results show that our maintenance algorithms are highly efficient. In particular, our ICD-order-based IC maintenance algorithms under edge insertion and edge deletion scenarios are up to six and three orders of magnitude faster than the SOTA IC maintenance methods, respectively. We have released the source codes and datasets of our work at: https://github.com/YouranSun/DynamicICS.

Contributions. In summary, our main contributions are:

- We analyze the limitations of all the existing index maintenance approaches for the IC search problem, and then theoretically prove that they are relatively unbounded, under both edge insertion and deletion scenarios.
- We propose a novel concept *ICD-order*, based on which we design efficient KC-Index maintenance algorithms under both edge insertion and deletion scenarios.
- We conduct extensive experiments on eight real-world large dynamic graphs, and the results show that our new algorithms are highly efficient and scalable.

Organization. We present the IC search problem in Section 2. Section 3 analyzes existing IC search works. We introduce the concept of *ICD-order* in Section 4, and present our KC-Index maintenance algorithms in Section 5. The experimental results are reported in Section 6. We review the related works in Section 7 and conclude in Section 8. For lack of space, all detailed proofs in this paper are included in our supplementary material.

2 Problem Definition

Table 1. Notations and meanings.

Notation	Meaning
$G = (V, E, \omega)$	an undirected weighted graph with vertex set V, edge set E, and weight vector ω
$G[S]$	the subgraph of G induced by vertex set S
G_k	the subgraph of the k-core in the graph G
$\mathcal{D}_k(G)$	the keynode set of G_k
$csv_k[v]$	the set of csv vertices corresponding to the v in G_k
O_k	the ICD-order of G_k
$N(v, G)$	the neighbor set of vertex v in G
$N_{O_k}(v, G)$	the remaining neighbor set of vertex v in G under the ICD-order O_k
G_k^+, G_k^-	G_k after an edge insertion and deletion respectively
$\mathrm{vol}(S)$	the sum of degrees in G of vertices $u \in S$

We consider a *vertex-weighted* undirected graph G=(V, E, ω), where V is the set of vertices, E is the set of edges, and ω is a weight vector that assigns each vertex $v \in V$ a weight denoted by $\omega(v)$. Here, the weight $\omega(v)$ represents the importance value of vertex v, which can be its PageRank

value [6], centrality score [38], h-index [5], or social status, etc. Let $n = |V|$ and $m = |E|$ denote the number of vertices and edges in G, respectively. Note that when vertex weights are unavailable, unsupervised influence estimation methods such as PageRank, or betweenness can be applied. If only partial weights are known, machine learning models like GNNs [34] or GATs [69] can estimate the missing ones. In addition, when multiple nodes share the same weight, tie-breaking strategies (e.g., node IDs or secondary attributes such as citation counts) ensure comparability. Following prior works [3, 12, 39, 40, 42, 49, 56, 73, 81], we assume that the weights of vertices are pre-given, and each vertex has a distinct weight (i.e., $\omega(u) \neq \omega(v), \forall u \neq v$). For each vertex $v \in G$, we use $N(v, G)$ to denote its set of neighbors in G. Given a vertex set $S \subseteq V$, we use $G[S] = (S, E(S), \omega)$ to denote the subgraph of G induced by S, where $E(S) = \{(u, v) \in E \mid u, v \in S\}$, and we let $N(S, G) = \bigcup_{v \in S} N(v, G)$. For any graph H, we denote its vertex set by $V(H)$ and its edge set by $E(H)$.

Definition 2.1 (k-core [1, 35]). Given an undirected graph G and an integer k ($k > 0$), the k-core is the maximal subgraph of G, such that each vertex in it has a degree of at least k.

The core number of a vertex $v \in V$ is defined as the largest k for which v is included in a non-empty k-core, denoted as $c(v)$. The maximum core number across all vertices in G is represented as δ, i.e., $\delta = \max\{c(u) \mid u \in V\}$. We denote by $G_k = (V_k, E_k, \omega)$ the subgraph of the k-core in the graph G.

Definition 2.2 (Influence value [3, 12, 40, 42, 49, 56, 73]). Given a subgraph $H = (V(H), E(H), \omega)$ of G, the influence value of H, denoted by $f(H)$, is defined as the minimum weight of the vertices in H, i.e., $f(H) = \min_{v \in V(H)} \omega(v)$.

Most existing influential community search studies [3, 40, 41, 46] adopt the $\min(\cdot)$ function, emphasizing that every member should be highly influential [40]. Recent work [56] further explores other aggregation functions (e.g., $\max(\cdot)$, $\text{avg}(\cdot)$, $\text{sum}(\cdot)$), showing that techniques for $min(\cdot)$ can be extended to these variants.

Definition 2.3 (Influential community (IC) [3, 12, 40, 42, 49, 56, 73]). Given a graph G and an integer k, a k-influential community (k-IC) is a vertex-induced subgraph H of G satisfying constraints:

- **Connectivity**. H is a connected k-core.
- **Maximality**. There is no other subgraph H' of G such that H' is a supergraph of H with $f(H') = f(H)$, and H' satisfies the connectivity constraint above.

In cases without ambiguity, we simply refer to a k-IC by the set of vertices that it contains.

In this paper, we study the problem of how to efficiently search ICs over large dynamic graphs. Although the vertices, edges, and weights can be changed, a vertex insertion (resp. deletion) can be transformed to a sequence of edge insertions (resp. deletions), and similarly, the changes of vertex weights can be modeled as a sequence of vertex updates [42]. Therefore, we focus mainly on developing novel efficient algorithms to search ICs when a new edge is inserted or an existing edge is deleted. We formally present our studied problem as follows, which follows the definition of existing IC search problem [3, 12, 40, 42, 49, 56, 73] except that we focus on dynamic graphs.

Problem 1. *Given a dynamic graph G, two integers k, and r, find the top-r k-ICs with the highest influence values after inserting a new edge or deleting an existing edge from G.*

For example, consider the static graph G of Figure 1 with k=3 and r=2. The two ICs are $\{v_6, v_7, v_9, v_{10}, v_{11}\}$ and $\{v_2, v_3, v_4, v_5\}$. After inserting a new edge (v_1, v_5), a new member v_1 joins the second IC, resulting in an updated IC of $\{v_1, v_2, v_3, v_4, v_5\}$.

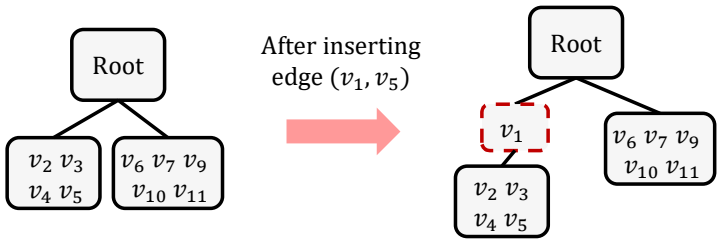


Fig. 3. The ICP-Index of the graph in Figure 1 for $k = 3$ before and after the edge insertion.

3 Analysis of existing works

In this section, we begin by presenting the IC decomposition (ICD) process, upon which two index-based algorithms are built. We then review these algorithms and discuss their limitations in dynamic settings.

3.1 IC decomposition

The process of computing all k-ICs, known as IC decomposition (ICD), involves iteratively removing the vertices with the smallest weights, followed by removing vertices with degrees less than k until all communities are identified. After this process, the vertices in the graph are classified into two categories: (1) keynodes, whose weights represent the influence values of the communities, and (2) *cvs*, which represent the remaining vertices [3]. Their definitions are as follows:

Definition 3.1 (Keynode [3]). A vertex u in a graph G_k is a keynode if there exists a k-core g of G_k such that g has an influence value $\omega(u)$ and contains the vertex u. Let $\mathcal{D}_k$ denote the set of keynodes in graph G_k.

Definition 3.2 (cvs vertex [3]). All vertices that are not keynodes in graph G_k are called the *cvs* vertices of G_k.

Given a graph G_k, let cvs_k denote the set of *cvs* vertices in G_k, and for each keynode $v \in \mathcal{D}_k$. We denote $cvs_k[v] = \{u \in V_k \mid \mathcal{D}_k[u] = v\}$ as the set of all *cvs* vertices corresponding to the keynode v, where $\mathcal{D}_k[u]$ is the keynode of u. A vertex u's keynode is defined as the vertex with the minimum weight among all ICs containing u. Thus, each *cvs* vertex uniquely corresponds to exactly one keynode.

Example 1. *Consider the graph in Figure 1 with $k = 3$, the keynodes in this graph are v_2 and v_6; their corresponding cvs vertices are $\{v_3, v_4, v_5\}$ and $\{v_7, v_9, v_{10}, v_{11}\}$, respectively, as shown in Figure 2. Therefore, there are two 3-ICs induced by $\{v_6, v_7, v_{10}, v_9, v_{11}\}$ and $\{v_2, v_3, v_4, v_5\}$ respectively.*

3.2 Review of ICP-Index and KC-Index

In this subsection, we review the two indices for efficient IC searching and then discuss their respective maintenance algorithms.

Overview of ICP-Index. Li et al. [40] proposed an index-based algorithm called ICP-Index that organizes ICs for a given k into a tree structure, with the entire index forming a forest. Specifically, because ICs are nested, they can be represented as a tree in which each node stores the distinct vertices not included in its child nodes. We denote the tree corresponding to a specific k as $\mathcal{T}_k$, where $1 \le k \le \delta$. For example, let k=3, the ICP-Index of G in Figure 1 is depicted in Figure 3, where the vertices in the nodes of each subtree compose the vertex set of a 3-IC.

Overview of KC-Index. In contrast, Bi et al. [3] developed a lightweight index that stores only the keynodes and their corresponding *cvs* vertices, called KC-Index, as shown in Figure 2. That is, KC-Index can be easily constructed after ICD, since it simply maintains the mapping between each keynode and its *cvs* vertices.

Relationship between ICP-Index and KC-Index. KC-Index can be regarded as a sequential version of ICP-Index, storing only the mapping between each keynode and its associated *cvs* vertices. In contrast, ICP-Index maintains a tree structure, where each tree node consists of a keynode in an IC (the minimum-weight vertex in the subtree) and its respective *cvs* vertices. Both indexes support efficient top-r IC search: ICP-Index locates the top-r subtrees (ICs), while KC-Index directly retrieves the top-r keynodes and their associated *cvs* vertices as induced subgraphs. Both methods achieve this in linear time. Besides, IC search based on the two indices shows comparable performance, and in some cases, the search based on KC-Index is even faster [3].

Algorithm 1: ICPIns [42]

input : A graph G_k, an edge $e = (u, v)$
output: The updated ICP-Index

1 // Suppose $\omega(\mathcal{D}_k[u]) \leq \omega(\mathcal{D}_k[v])$
2 update the core numbers for each vertex in G_k;
3 $\widetilde{c} \leftarrow \min\{c(u), c(v)\}$;
4 **for** $k = 1$ *to* $\widetilde{c}$ **do**
5 $\quad \mathcal{S} \leftarrow cvs[\mathcal{D}_k[u]] \cup \{\mathcal{D}_k[u]\}$;
6 $\quad$ **foreach** $w \in \mathcal{S}$ **do** recompute $\mathcal{L}_k(w)$;
7 $\quad Q = \{\mathcal{D}_k[u]\}$;
8 $\quad$ **while** $Q \neq \emptyset$ **do**
9 $\quad\quad w \leftarrow Q$.poll();
10 $\quad\quad$ **foreach** $x \in N(w, \mathcal{S})$ **do**
11 $\quad\quad\quad$ **if** $\mathcal{L}_k(x) = k$ **then** Q.add(x); ;
12 $\quad\quad\quad \mathcal{L}_k(x) \leftarrow \mathcal{L}_k(x) - 1$;
13 $\quad\quad$ delete w from Q and S;
14 $\quad$ **if** u *is not deleted from* S **then** recompute $\mathcal{T}_i$;
15 **return** $\mathcal{T}_1 \cdots \mathcal{T}_k$;

Maintain the ICP-Index. Utilizing the ICP-Index, Li et al. [42] further designed algorithms to maintain the ICP-Index under the edge insertion and deletion scenarios, denoted by ICPIns and ICPDel respectively. The main idea of these algorithms is to identify and rebuild the affected trees, which is based on the following definition:

Definition 3.3 (Layer degree [40]). Given a graph G, an integer k, and a keynode set $\mathcal{D}_k$, the layer degree of vertex v is defined as: $\mathcal{L}_k(v) = |\{u \mid u \in N(v, G_k) \wedge \omega(\mathcal{D}_k[u]) \geq \omega(\mathcal{D}_k[v])\}|$.

For the scenario of edge insertion, the ICPIns algorithm is shown in Algorithm 1. Specifically, when inserting a new edge (u, v) into the graph G_k, it first updates the core number of vertices in G_k, and only the trees with k values less than or equal to $\min\{c(u), c(v)\}$ need to be processed (lines 2-3). Each tree is handled sequentially (lines 3-14), for each tree $\mathcal{T}_k$ (suppose $\omega(\mathcal{D}_k[u]) \leq \omega(\mathcal{D}_k[v])$), it selects the tree node $\mathcal{S}$ that contains u and recomputes the layer degrees for all vertices in this node (lines 5-6). Then, it first deletes the vertex with the smallest weight in this tree node (i.e., $\mathcal{D}_k[u]$), decreases the layer degrees of its neighbors, and then iteratively removes all vertices with layer degrees less than k (lines 7-13). Afterward, if u remains in $\mathcal{S}$, then $\mathcal{T}_i$ needs to be rebuilt; otherwise,

Table 2. Concepts and meanings in the boundedness analysis.

Concept	Meaning
D	the index construction algorithm during the ICD process
M	the incremental algorithm to compute ICs, i.e., ICPIns, ICPDel, and our methods
$Q(G)$	the query for IC decomposition in a graph G
AFF	the cost difference between applying IC decomposition on G and $(G \oplus \Delta G)$.

it moves to handle the indexes for the subsequent k (line 14). Finally, the updated ICP-Index is returned (line 15).

For the scenario of edge deletion, it follows similar steps as ICPIns, when deleting an edge (u, v) from the graph, it processes the index one by one with some pruning.

Maintain the KC-Index. To the best of our knowledge, there are no existing works for maintaining the KC-Index. A naive idea is to first use a state-of-the-art k-core maintenance algorithm [80] to update the k-core of the graph, and then rebuild the KC-Index on the updated core. We denote this approach as KCIns and KCDel for edge insertion and deletion scenarios, respectively.

3.3 Detailed analysis of ICPIns and ICPDel

In this subsection, we first present the time complexities of ICPIns and ICPDel, introduce some useful definitions for analyzing the boundedness of the above algorithms, and then prove that they are all relatively unbounded.

Theorem 3.4 ([40]). *The total time cost of* ICPIns *for inserting an edge* (u, v) *from a graph* G *is* $O(\sum_{k=1}^{\psi} |E_k| + |V_k| \log |V_k|)$, *where* $\psi = \min\{c(u), c(v)\} + 1$. *The total time cost of* ICPDel *for deleting an edge* (u, v) *from a graph* G *is* $O(\sum_{k=1}^{\psi} |E_k| + |V_k| \log |V_k|)$.

If ΔG (e.g., an edge insertion or deletion) is the input update to G, applying ΔG to G results in the updated query $Q(G \oplus \Delta G)$, where $G \oplus \Delta G$ means the updated G by applying ΔG to G. Let $D(G \oplus \Delta G)$ represent computing $Q(G \oplus \Delta G)$ from scratch using algorithm D, and $M(G, \Delta G)$ denote using algorithm M to incrementally compute $Q(G \oplus \Delta G)$.

Some prior works introduced the concept of relative boundedness by defining the affected part processed by D, denoted as AFF.

*Definition 3.5 (*AFF *[20]).* Given a graph G, a query Q, and a set of edges ΔG updated to G, AFF signifies the cost difference of D between computing $Q(G)$ and $Q(G \oplus \Delta G)$.

For better clarity, please refer to Table 2 for the relevant definitions used in the context of IC search over dynamic graphs.

AFF accurately represents the cost required for incrementalizing D. If M is the incremental algorithm that incurs the minimum cost with respect to AFF, then M is considered a relatively bounded incremental algorithm for D. This leads to the following definition of relative boundedness.

Definition 3.6 (Relative boundedness [79]). An incremental graph algorithm M for Q is relatively bounded to D if its cost is polynomial in $|Q|$ and AFF.

Now, we are ready to prove that both ICPIns and ICPDel are relatively unbounded:

Theorem 3.7. ICPIns *and* ICPDel *are relatively unbounded algorithms.*

Table 3. The ICD-order of graph G in Figure 1.

Order	Vertices
O_1	$v_1 \prec v_2 \prec v_3 \prec v_4 \prec v_5 \prec v_6 \prec v_7 \prec v_8 \prec v_9 \prec v_{10} \prec v_{11}$
O_2	$v_1 \prec v_2 \prec v_3 \prec v_4 \prec v_5 \prec v_8 \prec v_6 \prec v_7 \prec v_{10} \prec v_9 \prec v_{11}$
O_3	$v_2 \prec v_3 \prec v_4 \prec v_5 \prec v_6 \prec v_7 \prec v_{10} \prec v_9 \prec v_{11}$

Proof sketch. To demonstrate the unboundedness of ICPIns and ICPDel, we first define the affected set AFF for IC decomposition augmented with the ICP-Index as auxiliary information. We then establish that the computational cost of ICPIns and ICPDel cannot be bounded by any polynomial in |AFF|. □

3.4 Opportunities

Based on the above analysis, we realize that there are two major limitations to the SOTA index maintenance algorithm. (1) *Theoretical guarantee.* As we reviewed before, both of ICPIns and ICPDel are relatively unbounded. (2) *Practical performance.* They still involve significant redundant computation, as a tree typically experiences only small changes rather than requiring a complete recomputation. That is, iterating over all vertices and edges in the graph to recompute trees each time they change becomes costly, particularly for large-scale graphs. For instance, on the as-skitter dataset with 10 million edges, they fail to maintain the ICP-Index for inserting or deleting 6,000 edges within three days. On the other hand, KC-Index, as a lightweight version of ICP-Index, enables more efficient maintenance due to its simple structure. However, such efficient algorithms for maintaining KC-Index under edge insertions and deletions have not been explored in the literature. *In this paper, we aim to design efficient algorithms for maintaining* KC-Index *in both edge insertion and deletion scenarios.*

4 Influential community decomposition order

In this section, we introduce a novel concept called ICD-order, which serves as a cornerstone of our approach. We then formulate diff for theoretical analysis.

Recall that to find the k-ICs, we need to first obtain the keynodes and their associated *cvs* vertices, where the ICD-order represents the sequence in which vertices are removed during this process. This sequence denoted as $O_k = \{v_1, v_2, \cdots, v_{|V_k|}\}$. If a vertex u is removed before v in this calculation process, it is denoted as $u \prec v$. For two vertices $v_i, v_j \in O_k$ with $i \neq j$, we call $v_i \prec v_j$, if $i < j$ and, $u \preceq v$ if $u \prec v$ or $u = v$. Therefore, for all possible values of k, we derive a set of sequences forming the ICD-order for the entire graph, denoted by $O_G = \{O_1, O_2, \cdots, O_\delta\}$, where δ denotes the maximum core number of G.

Example 2. *Let us continue with the graph G from Example 1, where the ICD-order of G is shown in Table 3, and the keynodes of O are marked in red color. Here,* $V(G_3) = \{v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}\}$, *and* v_2 *is the first vertex to be deleted in the ICD process, as it has the smallest weight among all vertices in* G_3. *After removing* v_2, *then* v_3, v_4, v_5 *are also deleted since their degrees are less than 3, due to the removal of* v_2. *Similarly,* v_6, v_7, v_{10}, v_9, *and* v_{11} *are removed sequentially. Thus, we obtain* O_3 *of* G_3.

Note that there are two types of vertices in the sequence of vertices O_k, where the vertex's "position" in O_k can reflect its type, and based on which, we can also find the keynodes and their corresponding *cvs* vertices. Here, we first introduce some novel definitions as follows:

Definition 4.1 (Remaining neighbor). Given a graph G_k=(V_k, E_k), and a vertex order O_k of V_k, for a vertex $u \in V_k$, the remaining neighbors of u, denoted as $N_{O_k}(u, G_k)$, are defined as the neighbors that have larger orders than u in O_k:

$$N_{O_k}(u, G_k) = \{w \mid w \in N(u, G_k) \wedge u \preceq w\}.$$

We use $d_{O_k}(u, G_k)$ to denote the remaining degree of u, i.e., $d_{O_k}(u, G_k) = |N_{O_k}(u, G_k)|$.

Example 3. *Consider the graph in Figure 1, we have $d_{O_3}(v_4, G_3) = 1$, since $N(v_4, G_3) = \{v_2, v_3, v_5\}$, and $v_2 \prec v_3 \prec v_4 \prec v_5$.*

Now we are ready to show how to identify the two types of vertices in the given ICD-order O_k.

Lemma 4.2. *Given a graph $G_k = (V_k, E_k)$ and its ICD-order O_k, a vertex $u \in O_k$ is a keynode of G_k, if and only if, $d_{O_k}(u, G_k) \geq k$.*

In ICD-order O_k, for two keynodes $u, v \in O_k$, if $\omega(u) < \omega(v)$, we have $u \preceq v$ in the O_k. That is, for two consecutive keynodes v_i and v_j ($i < j$) in O_k, the set of vertices that are positioned between v_i and v_j represents the $cvs_k[v_i]$ in O_k. Thus, we can easily obtain the `KC-Index` from the ICD-order O_k.

On the other hand, the `KC-Index` maintenance problem can be considered as the ICD-order maintenance problem. Specifically, let O be the ICD-order of a graph G, and O^+ (resp. O^-) denotes the ICD-order of the updated graph $G \oplus \Delta G$ (resp. $G \ominus \Delta G$). If we know the difference between O and O^+ (resp. O^-), we can derive the ICD-order of the updated graph efficiently, without recomputing it from scratch. To simplify the description, we only discuss the scenario of edge insertion later, and the edge deletion shares the same key idea as the insertion case. We quantify their difference by examining the ICD-orders of all graphs G_k=(V_k, E_k) of G and G_k^+=(V_k^+, E_k^+) of $G \oplus \Delta G$, for k from 1 to δ^+, where δ^+ denotes the maximum core number of $G \oplus \Delta G$. We use $\texttt{diff}_k(O_k, O_k^+)$ to denote the difference between O_k and O_k^+, where O_k and O_k^+ represents the ICD-orders of G_k and G_k^+, respectively. Formally, we have $\texttt{diff}_k(O_k, O_k^+) = \left\{w \in V_k \cap V_k^+ \mid \exists w' \in V_k \cap V_k^+, w \prec w' \wedge w' \prec^+ w\right\} \cup \left(V_k^+ \setminus V_k\right)$, where $w' \prec^+ w$ denotes that w' appear before w in the new ICD-order O_k^+.

Here, in other words, $\texttt{diff}_k(O_k, O_k^+)$ includes vertices whose positions in O_k^+ have shifted backward relative to their positions in the original order O_k. In summary, we have $\texttt{diff}(O, O^+) = \left(\cup_{k=1}^{\delta^+} \texttt{diff}_k(O_k, O_k^+)\right)$. Given a graph G and its ICD-order O, the new ICD-order O^+ of $G \oplus \Delta G$ may not be unique, while different orders all correspond exactly to the correct `KC-Index`. In this paper, we follow the existing works [80], and only consider the new ICD-order that has the minimum difference with O, i.e., $|\texttt{diff}(O, O^+)|$ is the smallest among all possible new ICD-orders.

Remark. The ICD-order represents the vertex removal sequence during the ICD process. That is, `diff` captures the difference between two successive ICD-orders caused by ΔG. It follows that $|\texttt{AFF}| = \sum_{k=1}^{\psi} \text{vol}(\texttt{diff}_k) + |N(\texttt{diff}_k, G_k)|$.

We present the following property of the ICD-order of G_k, which must satisfy the following three key conditions.

Property 1. *Given a graph G_k=(V_k, E_k), a sequence of vertices $O_k = \{v_1, v_2, \cdots, v_{|V_k|}\}$ qualifies for the ICD-order of G_k, if and only if the following three conditions are satisfied:*

- ***Condition 1.*** *For each keynode $v \in O_k, d_{O_k}(v, G_k) \geq k$, and for each vertex $w \in O_k, v \preceq w$, we have $\omega(v) < \omega(w)$. Intuitively, all keynodes in an ICD-order are arranged in ascending*

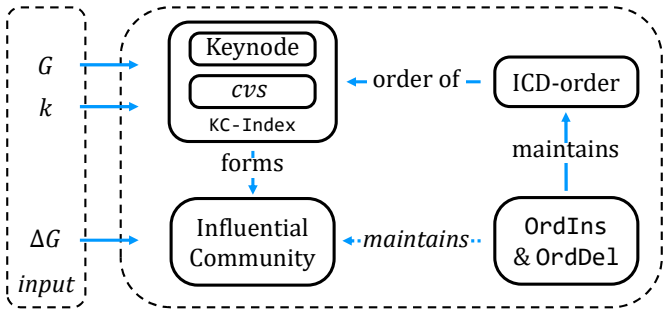


Fig. 4. The framework of our algorithms.

order of their weights. That is, if a keynode u appears after a keynode v in the ICD-order, then $\omega(u) > \omega(v)$.

- ***Condition 2.*** *Given a keynode $v \in O_k$, for each vertex $w \in O_k$ such that $v \preceq w$, we have $|N(w, G_k) \cap \{u \mid v \preceq u \wedge u \in O_k\}| \geq k$. Intuitively, for any keynode v, the subgraph induced by all vertices with order larger than v is a k-core, i.e., every vertex in it has degree at least k.*
- ***Condition 3.*** *For each cvs $v \in O_k$, we have $d_{O_k}(v, G_k) < k$. Intuitively, given an ICD-order, the remaining degree of each cvs vertex is less than k.*

The above property serves as the foundation for ensuring the correctness of our algorithms. In the remainder of this paper, we focus on designing efficient algorithms for maintaining the ICD-order in dynamic graphs. In other words, maintaining the ICD-order is equivalent to maintaining the `KC-Index` [3]. Thus, ICS queries over dynamic graphs can be efficiently answered by tracking changes to the ICD-order. The overall workflow of our algorithms is illustrated in Figure 4.

5 ICD-Order maintenance algorithms

In this section, we present efficient algorithms for maintaining the ICD-order under the scenarios of edge insertion and deletion.

5.1 ICD-order maintenance under the edge insertion scenario

In this subsection, we present the algorithm for efficiently maintaining the ICD-order tailored to handle edge insertion.

Upon inserting a new edge (u, v) into G_k, the updated graph is denoted as G_k^+. W.l.o.g., we assume $\omega(u) < \omega(v)$, and due to the edge insertion, some new vertices whose core numbers increased by one would be added to G_k from G_{k-1}, which is denoted by $\Delta V = V(G_k^+) \setminus V(G_k)$. That is, we need to insert ΔV into O_k, and find the correct positions for vertices in $O_k \cup \Delta V$ to obtain the new ICD-order O_k^+ of G_k^+, where the correct position of a vertex v refers to its position in the ICD-order O_k^+.

Key idea of order maintenance. Intuitively, our algorithm iteratively places the keynodes in their correct positions within the new ICD-order O_k^+, following an ascending order of their weights. As each keynode is placed, its corresponding *cvs* vertices are also placed in the new order. This process progressively expands *the set of vertices with correct positions* in O_k^+, ensuring that vertices added later always have a larger order than those added earlier. To achieve this, we maintain three disjoint vertex sets $\mathcal{R}$, Ψ, and $\mathcal{P}$, where their respective meanings are explained as follows:

- **Relative set** $\mathcal{R}$. The set of vertices that keep the same relative positions as each of them is in the original order, while their positions in the new ICD order are not yet determined.

- **Order set** Ψ. The first $|\Psi|$ vertices in the new ICD-order.
- **Candidate set** $\mathcal{P}$. The set of vertices whose positions have shifted backward relative to their positions in the original order or vertices that are newly added to O_k.

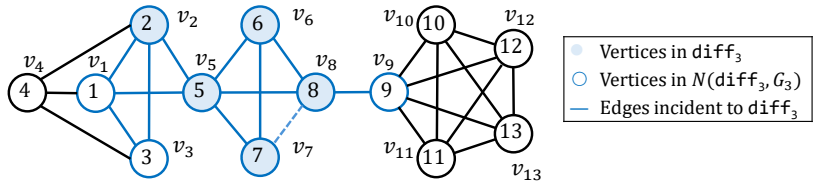
Based on this, the ICD-order maintenance process can be considered as a series of state transformation processes, starting from the state $\mathcal{S}_0 = (\mathcal{R}_0 = O_k, \Psi_0 = \emptyset, \mathcal{P}_0 = \Delta V)$ and ending at the state $\mathcal{S}_t = (\mathcal{R}_t = \emptyset, \Psi_t = O_k^+, \mathcal{P}_t = \emptyset)$, where t denotes the total number of steps. That is, the key design of our algorithm is that, in the i-th state, we determine which vertices in $\mathcal{P}_{i-1}$ and $\mathcal{R}_{i-1}$ can be appended to Ψ_i, and accordingly update these two sets for the next state. After t states, we can thus obtain the new ICD-order. In the remainder of this subsection, we first introduce the detailed process of state transformation, then we provide a concrete example to illustrate how the state is transformed. Finally, we present our order-based algorithm for maintaining the ICD-order under the edge insertion scenario.

State transformation process. In the i-th state, we yet do not know the final vertex order of $\mathcal{R}_i \cup \mathcal{P}_i$ in O_k^+. While we observe that the positions of $\mathcal{P}_i$ and their neighbors in $\mathcal{R}_i$ are mutually affected in the new order, since the ICD-order is constructed by iteratively removing vertices and their neighbors. Based on this, in the i-th state, we identify the largest prefix vertex sequence in $\mathcal{R}_i$ whose positions remain unaffected by the vertices in $\mathcal{P}_i$. Clearly, these vertices can then be directly appended to Ψ_i. In other words, in the i-th state, we can identify a sequence of vertices whose final orders in O_k^+ can be determined. Thus, our task becomes identifying the endpoint of this prefix sequence. To do so, we locate the neighbor of $\mathcal{P}_i$ in $\mathcal{R}_i$ with the smallest order and refer to this vertex as the "navigation node ", denoted by $\pi(\mathcal{P}_i)$. This node serves as a potential endpoint of the prefix sequence.

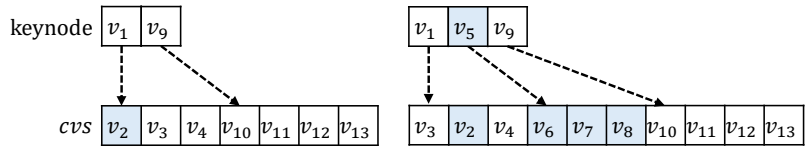
To check whether $\pi(\mathcal{P}_i)$ is a valid endpoint, we need to compare $\omega(p^*)$ with $\omega(\mathcal{D}_k[\pi(\mathcal{P}_i)])$, where p^* is the vertex with the minimum weight in $\mathcal{P}_i$. If $\omega(p^*) > \omega(\mathcal{D}_k[\pi(\mathcal{P}_i)])$, then $\pi(\mathcal{P}_i)$ is a valid endpoint. Otherwise, p^* is the endpoint. By such comparsion, we can identiy the vertex squence can be appended into the Ψ_i, that is, finsihing the state transformation of this state.

Here, we quickly explain why this comparison is needed: Note that if $\omega(p^*) > \omega(\mathcal{D}_k[\pi(\mathcal{P}_i)])$, then all vertices in $\mathcal{P}_i$ belong to either the keynode $\mathcal{D}_k[\pi(\mathcal{P}_i)])$ or keynodes that appear after $\mathcal{D}_k[\pi(\mathcal{P}_i)]$ in the new order. Thus, $\pi(\mathcal{P}_i)$ is a valid endpoint. On the other hand, if $\omega(p^*) < \omega(\mathcal{D}_k[\pi(\mathcal{P}_i)])$, then p^* must be a new keynode with a smaller order than $\pi(\mathcal{P}_i)$ in this new order. This is because we can easily verify that the remaining degree of p^* in the new order is larger than k. In this case, we select p^* as the endpoint, and all keynodes in $\mathcal{R}_i$ with smaller weights than p^* and their *cvs* vertices can be directly appended to Ψ_i.

We note that when $\pi(\mathcal{P}_i)$ is a valid endpoint, if its type and position remain unchanged in the new order, then $\pi(\mathcal{P}_i)$ can also be appended to the end of Ψ_i. This is because, in such cases, all vertices in $\mathcal{P}_i$ will still appear after it in the new order. To verify this, we consider two possible types of $\pi(\mathcal{P}_i)$: *(1)* $\pi(\mathcal{P}_i)$ *is a keynode in* O_k*:* its type and position must remain unchanged in the new order. *(2)* $\pi(\mathcal{P}_i)$ *is a cvs vertex in* O_k*:* verifying whether it still qualifies as a *cvs* vertex and maintains the same order in the new ICD-order is non-trivial, since its exact remaining degree in the updated ICD-order cannot be computed at this stage. To achieve this, we design a novel method to compute the upper bound of the remaining degree for vertex $u \in \mathcal{P}_i \cup \mathcal{R}_i$, by assuming that all vertices in $\mathcal{P}_i$ are placed after u in the new ICD order, which is denoted as the supremum degree $\Gamma(u)$ of u:



(a) A graph G with edge (v_7, v_8) to be inserted



(b) The keynodes and cvs of G (c) The keynodes and cvs of $G \oplus \Delta G$

Fig. 5. A toy example graph with edge insertion.

Definition 5.1 (Supremum degree). Upon edge insertion, in the i-th state of the order maintenance process, the supremum degree of a vertex $u \in \mathcal{R}_i \cup \mathcal{P}_i$, $\Gamma(u)$ is defined as:

$$\Gamma(u) = \begin{cases} \left| N_{O_k}\left(u, G_k^+[\mathcal{R}_i]\right) \cup N\left(u, G_k^+[\mathcal{P}_i]\right) \right|, & u \in \mathcal{R}_i \\ \left| N\left(u, G_k^+[\mathcal{R}_i]\right) \cup N\left(u, G_k^+[\mathcal{P}_i]\right) \right|, & u \in \mathcal{P}_i \end{cases}$$

Thanks to the above novel definition, we can conclude that if the supremum degree of $\pi(\mathcal{P}_i)$ is less than k, it must be a cvs vertex in O_k^+, and can be appended into the end of Ψ_i. Otherwise, if $\Gamma(\pi(\mathcal{P}_i)) \geq k$, it undergoes a type or position change in the new order. We append it to $\mathcal{P}_i$ and move to the next state (i.e., the $(i+1)$-th state).

Based on the above discussions, in the i-th state, we first select navigation node $\pi(\mathcal{P}_i)$ from $\mathcal{R}_i$ and then compare $\omega(\mathcal{D}_k[\pi(\mathcal{P}_i)])$ and $\omega(p^*)$. Based on their relationship, two cases need to be considered.

- **Case a.** If $\omega(p^*) < \omega(\mathcal{D}_k[\pi(\mathcal{P}_i)])$, p^* is a new keynode in the new ICD-order. The cvs vertices of p^* in $\mathcal{P}_i$ can be computed by: removing p^* from $\mathcal{P}_i$, and then all vertices in $\mathcal{P}_i$ with supreme degrees less than k are iteratively removed and appended into the cvs vertices of p^*.
- **Case b.** If $\omega(p^*) > \omega(\mathcal{D}_k[\pi(\mathcal{P}_i)])$, we need to check whether $\pi(\mathcal{P}_i)$ is a keynode, a cvs, or changes its type in the new order:
  - **Case b(i).** If $\pi(\mathcal{P}_i)$ is a keynode of O_k, then $\pi(\mathcal{P}_i)$ is also a keynode in O_k^+. The cvs vertices of $\pi(\mathcal{P}_i)$ in $\mathcal{P}_i$ can be computed similarly to **case a**. Then, we remove $\pi(\mathcal{P}_i)$ from $\mathcal{R}_i$, and append $\pi(\mathcal{P}_i)$ along with the removed vertices from $\mathcal{P}_i$ to Ψ_i.
  - **Case b(ii).** If $\Gamma(\pi(\mathcal{P}_i)) < k$, $\pi(\mathcal{P}_i)$ is still a cvs vertex in new order O_k^+.
  - **Case b(iii).** Otherwise, the position of $\pi(\mathcal{P}_i)$ shifts backward in O_k^+, or $\pi(\mathcal{P}_i)$'s type changes in O_k^+. We add $\pi(\mathcal{P}_i)$ to $\mathcal{P}_i$ for the subsequent computation.

The above two cases determine whether $\pi(\mathcal{P}_i)$ and p^* can serve as the endpoints of the vertex sequence. Based on this, we can identify which vertices can be appended to Ψ_i and accordingly update the vertex sets $\mathcal{P}_i$ and $\mathcal{R}_i$. Next, we provide a concrete example to illustrate how the above cases are applied to perform the state transformation.

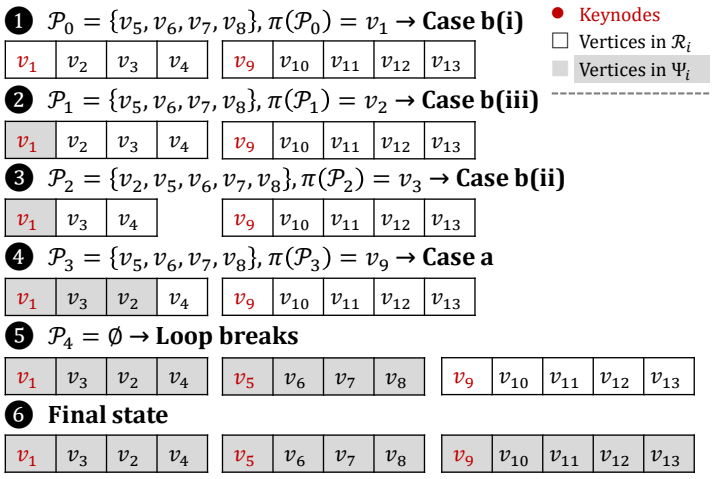


Fig. 6. The idea of order maintenance with edge insertion.

Building on the two cases above, inserting an edge into the graph correctly maintains the ICD-order, as shown in Figure 6. This figure illustrates how our ICD-order maintenance strategy handles edge insertion. Specifically, in the example from Figure 5 with $k = 3$, before inserting edge (v_7, v_8), the ICD-order $O_3 = \{v_1, v_2, v_3, v_4, v_9, v_{10}, v_{11}, v_{12}, v_{13}\}$. The vertices whose core number increases from 2 to 3 are $\{v_5, v_6, v_7, v_8\}$ (i.e., $\mathcal{P}_0$), with Ψ_0 and $\mathcal{R}_0$ initially empty. The following state transformation process inserts $\{v_5, v_6, v_7, v_8\}$ into O_3 and eventually obtains the new ICD-order O_3^+. ❶ $\mathcal{R}_0 = \{v_1, v_2, v_3, v_4, v_9, v_{10}, v_{11}, v_{12}, v_{13}\}$, $\Psi_0 = \emptyset$, and $\mathcal{P}_0 = \{v_5, v_6, v_7, v_8\}$, $\pi(\mathcal{P}_0) = v_1$, since $\omega(v_5) > \omega(\mathcal{D}_k[v_1])$ and v_1 is a keynode in O_k, it locates into **case b(i)**, where v_1 is still a keynode in O_3^+. In this case, there are no vertices in $\mathcal{R}_0$ with a smaller order than v_1, that is, v_1 can reserve its original order, so we directly append v_1 to Ψ_0. Then, the supremum degrees of each vertex in $\mathcal{P}_0$ are updated. Specifically, the supremum degrees of v_5, v_6, v_7, and v_8 are updated to 4, 3, 3, and 4, respectively, with no vertices removed from $\mathcal{P}_0$. ❷ $\mathcal{R}_1 = \{v_2, v_3, v_4, v_9, v_{10}, v_{11}, v_{12}, v_{13}\}$, $\Psi_1 = \{v_1\}$, and $\mathcal{P}_1 = \{v_5, v_6, v_7, v_8\}$, $p^* = v_5$ and $\pi(\mathcal{P}_1) = v_2$. Since $\omega(v_5) > \omega(\mathcal{D}_k[v_2])$ and $\Gamma(v_2) = 3$, it belongs to **case b(iii)** where v_2 cannot stay at the current position because it has three neighbors in all undetermined vertices in $\mathcal{R}_1 \cup \mathcal{P}_1$. That is, we move v_2 to $\mathcal{P}_1$, and update $\mathcal{R}_1$. ❸ $\mathcal{R}_2 = \{v_3, v_4, v_9, v_{10}, v_{11}, v_{12}, v_{13}\}$, $\Psi_2 = \{v_1\}$, and $\mathcal{P}_2 = \{v_2, v_5, v_6, v_7, v_8\}$, $p^* = v_2$, $\pi(\mathcal{P}_2) = v_3$. Since $\omega(v_2) > \omega(\mathcal{D}_k[v_3])$ and $\Gamma(v_3) = 2$, it falls under **case b(ii)**. The position of v_3 can be determined, and v_2 is added to the *cvs* vertices of v_1, immediately after v_3, since $\Gamma(v_2) < 3$. v_3 and v_2 are appended to Ψ_2. ❹ $\mathcal{R}_3 = \{v_4, v_9, v_{10}, v_{11}, v_{12}, v_{13}\}$, and $\Psi_3 = \{v_1, v_3, v_2\}$, $\mathcal{P}_3 = \{v_5, v_6, v_7, v_8\}$, $\pi(\mathcal{P}_3) = v_9$. As $\omega(v_5) < \omega(\mathcal{D}_k[v_9])$, it belongs to **case a** and v_6, v_7, v_8 are removed from $\mathcal{P}_3$ in order and appended to the *cvs* vertices of v_5. That is, we need to find a proper position of keynode v_5 and its *cvs* verticies $\{v_6, v_7, v_8\}$. As v_9 is the first keynode with weight greater than $\omega(v_5)$, vertices before v_9 (i.e., v_4) are be appended to Ψ_3, followed by v_5, v_6, v_7, v_8. ❺ $\mathcal{R}_4 = \{v_9, v_{10}, v_{11}, v_{12}, v_{13}\}$, $\Psi_4 = \{v_1, v_3, v_2, v_4, v_5, v_6, v_7, v_8\}$, and $\mathcal{P}_4 = \emptyset$. $\mathcal{P}_4$ becomes empty and the loop ends. Then, all remaining vertices in $\mathcal{R}_4$ should be directly appended to Ψ_4. ❻ Finally, $\mathcal{R}_5$ and $\mathcal{P}_5$ both become empty and the new ICD-order is $\Psi_5 = \{v_1, v_3, v_2, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}, v_{12}, v_{13}\}$. In Figure 5, we show the vertices in $\texttt{diff}_3$ and $N(\texttt{diff}_3, G_3)$, and the edges in $\text{vol}(\texttt{diff}_3)$.

5.2 Overall edge insertion algorithm

Based on the discussions above, we can present an efficient algorithm to maintain the ICD-order under the scenario of edge insertion, as shown in Algorithm 2.

Algorithm 2: `OrdIns`

input : A graph G, an edge $e = (u, v)$, the ICD-order $\mathcal{O}$ of G
output : The new ICD-order $\mathcal{O}^+$ of G^+

1 $G^+ \leftarrow G \oplus \{e\}$; $\mathcal{O}^+ \leftarrow \emptyset$;
2 **for** $k \leftarrow 1$ **to** $\min\{c(u), c(v)\} + 1$ **do**
3 $\mathcal{O}^+ \leftarrow \mathcal{O}^+ \cup$ `Order-InsK`$(k, G_k^+, e, \mathcal{O}_k)$;
4 **return** $\mathcal{O}^+$;

5 **Function** `Order-InsK`$(k, G_k^+, e = (u, v), \mathcal{O}_k)$**:**
6 $\mathcal{R}_0 \leftarrow \mathcal{O}_k$; $\mathcal{P}_0 \leftarrow V(G_k^+) \backslash V(G_k)$; $\Psi_0 \leftarrow \emptyset$; $i \leftarrow 0$;
7 **while** $\mathcal{P}_i \neq \emptyset$ **do**
8 $\pi(\mathcal{P}_i) \leftarrow$ the nagivation node of $\mathcal{P}_i$;
9 $p^* \leftarrow$ the vertex with minimum weight in $\mathcal{P}_i$;
10 $\mathcal{U}_i \leftarrow$ vertices in $\mathcal{R}_i$ that have smaller orders than $\pi(\mathcal{P}_i)$;
11 // Handle Case a: p^* is a new keynode
12 **if** $\omega(p^*) < \omega(\mathcal{D}_k[\pi(\mathcal{P}_i)])$ **then**
13 $cvs \leftarrow$ `DeleteVertex`$(p^*, \mathcal{P}_i, \mathcal{P}_i, G_k^+, k)$;
14 $kn \leftarrow$ the first keynode in $\mathcal{U}_i$ with a weight $> \omega(p^*)$;
15 insert all vertices in $\mathcal{U}_i$ with orders smaller than kn into Ψ_i, followed by p^* and its cvs vertices;
16 **else**
17 // Handle the Case b(i) and case b(ii).
18 insert all vertices of $\mathcal{U}_i$ into Ψ_i;
19 **if** $\pi(\mathcal{P}_i)$ *is a keynode* ***or*** $\Gamma(\pi(\mathcal{P}_i)) < k$ **then**
20 $cvs \leftarrow$ `DeleteVertex`$(\pi(\mathcal{P}_i), \mathcal{P}_i, \mathcal{R}_i, G_k^+, k)$;
21 insert $\pi(\mathcal{P}_i)$ and cvs into Ψ_i;
22 // Handle the case b(iii).
23 **else** insert $\pi(\mathcal{P}_i)$ into $\mathcal{P}_i$, and remove it from $\mathcal{R}_i$;
24 remove all vertices added to Ψ_i from $\mathcal{R}_i$ and $\mathcal{P}_i$; $i \leftarrow i + 1$;
25 append all remaining vertices in $\mathcal{R}_i$ to Ψ_i;
26 **return** Ψ_i;

Specifically, we initialize $\mathcal{O}^+ = \emptyset$ and then compute the new ICD-order $\mathcal{O}_k^+$ of G_k via function `Order-InsK`, for k from 1 to $\min\{c(u), c(v)\} + 1$ (lines 2–4). For a given k, `Order-InsK` maintains the updated ICD-order via a while loop (lines 7–24). In each state i, it selects a pivot vertex p^* from $\mathcal{P}_i$, computes the navigation node $\pi(\mathcal{P}_i)$, and determines the update set $\mathcal{U}_i$ (lines 8–9). If $\omega(p^*) < \omega(\mathcal{D}_k[\pi(\mathcal{P}_i)])$ (**case a**), p^* is a new keynode, and its corresponding cvs vertices can be computed by removing vertices with the remaining supremum degree less than k (line 12–13). We then locate the first keynode kn with weight greater than $\omega(p^*)$, and insert vertices in $\mathcal{U}_i$ with orders smaller than kn, followed by p^* and its cvs vertices into Ψ_i (lines 14–15). For **case b**, all $\mathcal{U}_i$ vertices are first appended to Ψ_i, and then we determine whether subcase **b(i)** or **b(ii)** applies (lines 18–19). If so, we compute the corresponding cvs vertices and append both $\pi(\mathcal{P}_i)$ and cvs of $\mathcal{D}_k[\pi(\mathcal{P}_i)]$ in $\mathcal{P}_i$ to Ψ_i (lines 20–21). In subcase **b(iii)**, $\pi(\mathcal{P}_i)$ is reinserted into $\mathcal{P}_i$ (line 23). At the end of each state, updated vertices are removed from $\mathcal{R}_i$ and $\mathcal{P}_i$, and the algorithm proceeds to the next state (line 24). Once $\mathcal{P}_i = \emptyset$, the remaining vertices in $\mathcal{R}_i$ are appended to Ψ_i, yielding the final

state $\Psi_t = O_k^+$, with $\mathcal{P}_t = \mathcal{R}_t = \emptyset$. Repeating this process for all k values yields the final ICD-order O^+.

In the following, we aim to prove the correctness of our ICD-order maintenance algorithm, `OrdIns`, by proving that in the last state, Ψ_t satisfies the properties of the ICD-order.

Theorem 5.2. *Given a graph G, a set of edges ΔG, and G's ICD-order O, Algorithm 2 can compute $G \oplus \Delta G$'s ICD-order O^+ correctly.*

Theorem 5.3. *The time complexity of Algorithm 2 for inserting an edge (u, v) is $O\big(\sum_{k=1}^{\psi}(\mathrm{vol}(\mathtt{diff}_k) + |N(\mathtt{diff}_k, G_k)| \cdot \log|V_k|)\big)$, where $\psi = \min\{c(u), c(v)\} + 1$, and $\mathtt{diff}_k$ denotes difference between ICD-orders O_k and O_k^+.*

Theorem 5.4. *Given a graph G, its ICD-order O, and a set of edges ΔG to be inserted into G, Algorithm 2 is relatively bounded with respect to the ICD maintenance algorithm.*

5.3 ICD-order maintenance under the edge deletion scenario

In this subsection, we present the algorithm for efficiently maintaining the ICD-order tailored to handle edge deletion.

Upon deleting an edge (u, v), the updated graph is denoted as G_k^-. W.l.o.g., we assume $\omega(u) < \omega(v)$, and due to the edge deletion, some vertices whose core numbers decrease by one would be deleted from G_k, which is denoted by $\Delta V = V(G_k) \backslash V(G_k^-)$. That is, we need to delete ΔV from O_k and also maintain the correct positions for the remaining vertices to obtain the new ICD-order O_k^- of G_k^-.

Key idea of order maintenance. The algorithm for edge deletion mirrors insertion, but with two critical adjustments: (1) it processes keynodes in descending order of weights (vs. ascending in insertion); (2) the set of determined-position vertices in O_k^- also grows progressively, but the newly added vertices are placed before earlier ones in the updated ICD-order. For edge deletion, we also maintain three disjoint vertex sets $\mathcal{R}$, Ψ, and $\mathcal{P}$, where $\mathcal{R}$ remains the same meaning as in edge insertion, Ψ denotes the last $|\Psi|$ vertices in the new ICD-order, and $\mathcal{P}$ represents the set of vertices whose positions have shifted forward relative to their original order or vertices that have been removed from O_k. The process starts at $\mathcal{S}_0 = (\mathcal{R}_0 = O_k, \Psi_0 = \emptyset, \mathcal{P}_0 = \Delta V)$ and terminates at $\mathcal{S}_t = (\mathcal{R}_t = \emptyset, \Psi_t = O_k^-, \mathcal{P}_t = \Delta V)$. Unlike the insertion algorithm, the vertices in ΔV do not enter O_k^-; instead, they only serve to guide the order adjustments of other vertices.

State transformation process. From state i to $i+1$, we first select the vertex with the largest order in $\mathcal{P}_i$, and then, we denote its keynode as the navigation node (i.e., $\pi(\mathcal{P}_i)$), which plays a similar role to the navigation node in edge insertions. It is because instead of considering each single vertex, here, we are processing a keynode and its *cvs* vertices at a time. Here, $\pi(\mathcal{P}_i)$ is the keynode with the largest weight that may change its position in $\mathcal{R}_i$. This implies that all vertices in $\mathcal{R}_i$ with larger orders than all *cvs* vertices of $\pi(\mathcal{P}_i)$ must retain their positions in the new order. Therefore, we can append them directly to the front of Ψ_i. As for $\pi(\mathcal{P}_i)$ and its *cvs* vertices, there are two possible cases: (1) $\pi(\mathcal{P}_i)$ is still a keynode and its *cvs* can also retain their positions. In addition, some vertices in $\mathcal{P}_i$ can also be identified as its *cvs* vertices. (2) $\pi(\mathcal{P}_i)$ becomes a *cvs* vertex in the new ICD-order. We should assign $\pi(\mathcal{P}_i)$ all its *cvs* vertices to new keynodes with smaller orders. Therefore, we need to check whether $\pi(\mathcal{P}_i)$ is still a keynode in the new order. In the i-th state, we also cannot know the remaining degree of $\pi(\mathcal{P}_i)$ in the new ICD-order in advance, thus we turn to design a novel strategy to compute its lower bound. Specifically, for each vertex $u \in \mathcal{R}_i$, we assume that all its neighbors in $\mathrm{cvs}_k[\pi(\mathcal{P}_i)]$ and $\mathcal{P}_i$ have larger orders than u in the new order. Here, for simplification, we use $\mathcal{S}_i$ to denote $\pi(\mathcal{P}_i)$ union its *cvs* vertices at the i-th state, which

serves as an auxiliary set to identify the vertex type of $\pi(\mathcal{P}_i)$. By doing this, the infimum degree of each vertex $u \in \mathcal{S}_i \cup \mathcal{P}_i$ can be computed by:

Definition 5.5 (Infimum degree). Upon edge deletion, in the i-th state of the order maintenance process, the infimum degree of a vertex $u \in \mathcal{S}_i \cup \mathcal{P}_i$ is defined as: $\beta(u) = |N_{O_k}(u, G_k^-[\mathcal{R}_i]) \cup N(u, G_k^-[\mathcal{P}_i \cup \mathcal{S}_i \cup \Psi_i])|$.

Based on this novel definition, all vertices in $\mathcal{S}_i \cup \mathcal{P}_i$ with infimum degrees less than k are iteratively removed, since these vertices must appear before $\pi(\mathcal{P}_i)$ in the new ICD-order. If the navigation node $\pi(\mathcal{P}_i)$ remains in $\mathcal{S}_i$ after this step, its infimum degree is at least k, meaning it is still a keynode in the new order. Excluding $\pi(\mathcal{P}_i)$, the remaining vertices in $\mathcal{S}_i \cup \mathcal{P}_i$ are the *cvs* vertices of $\pi(\mathcal{P}_i)$ in the new ICD-order, and then we can append $\pi(\mathcal{P}_i)$ along with its *cvs* vertices into the front of Ψ_i. Otherwise, $\pi(\mathcal{P}_i)$ is a *cvs* vertex in the new ICD-order. Here, all vertices in $\mathcal{S}_i \cup \mathcal{P}_i$ that are removed during the above process, including $\pi(\mathcal{P}_i)$, form the $\mathcal{P}_{i+1}$ vertex set in the next state.

Algorithm 3: `Order-DelK`

input : A graph G, an edge $e = (u, v)$, the ICD-order O of G
output: The new ICD-order O_k^- of G_k^-

1 $\mathcal{R}_0 \leftarrow O_k$; $\mathcal{P}_0 \leftarrow V(G) \backslash V(G^-)$; $\Psi_0 \leftarrow \emptyset$; $i \leftarrow 0$;
2 $\beta(u) \leftarrow \beta(u) - 1$;
3 **while** $\mathcal{P}_i \neq \emptyset$ **do**
4 $\pi(\mathcal{P}_i) \leftarrow$ the navigation node of $\mathcal{P}_i$;
5 $\mathcal{S}_i \leftarrow \{\pi(\mathcal{P}_i)\} \cup cvs_k[\pi(\mathcal{P}_i)]$;
6 $\mathcal{U}_i \leftarrow$ the vertices in $\mathcal{R}_i$ that have larger orders than all vertices in $\mathcal{S}_i$;
7 append all vertices in $\mathcal{U}_i$ to the front of Ψ_i;
8 $\mathcal{S}_i, \mathcal{P}_i, \mathcal{W}_i \leftarrow$ `CheckKeynode`$(\mathcal{S}_i, \mathcal{P}_i, G_k^-, k)$;
9 **if** $\pi(\mathcal{P}_i) \in \mathcal{S}_i$ **then**
10 // $\pi(\mathcal{P}_i)$ is still a keynode
11 **if** $\mathcal{S}_i \cup \mathcal{P}_i$ *is not same as* $(\{\pi(\mathcal{P}_i)\} \cup cvs_k[\pi(\mathcal{P}_i)])$ **then**
12 $cvs \leftarrow$`DeleteVertex`$(\pi(\mathcal{P}_i), \mathcal{S}_i \cup \mathcal{P}_i, \mathcal{R}_i, G_k^-, k)$;
13 **else** $cvs \leftarrow cvs_k[\pi(\mathcal{P}_i)]$;
14 append $\pi(\mathcal{P}_i)$ and its *cvs* vertices to the front of Ψ_i;
15 $\mathcal{P}_i \leftarrow \mathcal{W}_i$;
16 remove all vertices added to Ψ_i and $\mathcal{P}_i$ in this state from $\mathcal{R}_i$;
17 move to the next state by setting $i \leftarrow i + 1$;
18 append all vertices in $\mathcal{R}_i$ to the front of Ψ_i;
19 **return** Ψ_i;

Our edge deletion algorithm `OrdDel` follows a similar framework to the `OrdIns`. It needs to compute the new ICD-order O_k^- of G_k via function `Order-DelK`, for k from 1 to $\min\{c(u), c(v)\}$. We present the details of `Order-DelK` in Algorithm 3. For a given k, it maintains the updated ICD-order via a while loop (lines 3–17). In i-th state, it first computes the navigation node $\pi(\mathcal{P}_i)$, and calculates $\mathcal{S}_i$ according to $\pi(\mathcal{P}_i)$ and its *cvs* vertices (lines 4-5). Then, it computes $\mathcal{U}_i$ which contains all vertices in $\mathcal{R}_i$ that have larger orders than vertices in $\mathcal{S}_i$ and append them to Ψ_i (lines 6-7). Next, we update the infimum degrees of vertices in $\mathcal{S}_i \cup \mathcal{P}_i$, and iteratively remove those with infimum degrees less than k, storing them in $\mathcal{W}_i$ (line 8). The details of `CheckKeynode` are shown in our supplementary material. Then, if $\pi(\mathcal{P}_i) \in \mathcal{S}_i$, it is still a keynode in the O_k^-, and its

corresponding *cvs* vertices can be computed by iteratively removing the vertices with remaining infimum degrees less than k from $\mathcal{S}_i \cup \mathcal{P}_i$ (line 12). Afterwards, we append $\pi(\mathcal{P}_i)$ and its *cvs* vertices in the front of Ψ_i. Specifically, if the *cvs* vertices of $\pi(\mathcal{P}_i)$ remains the same as in O_k, we directly append its original *cvs* vertices (lines 13-14). For case b, all vertices in $\mathcal{S}_i \cup \mathcal{P}_i$ have been removed to $\mathcal{W}_i$. For both cases a and b, $\mathcal{P}_i$ is updated to $\mathcal{W}_i$ for the next state computing (line 15). At the end of each state, we remove the vertices that are added to Ψ_i from $\mathcal{R}_i$ and $\mathcal{P}_i$, correspondingly and move to the next state (lines 16-17). Until $\mathcal{P}_i$=$\emptyset$, we append all remaining vertices in $\mathcal{R}_i$ to the front of Ψ_i, and obtain the new ICD-order, which is the Ψ_i in the last state (i.e., Ψ_t) (lines 18-19).

Theorem 5.6. *Given a graph G with its ICD-order O, and a set of edges ΔG,* `OrdDel` *can compute $G \ominus \Delta G$'s ICD-order O^- correctly.*

Moreover, we measure the difference between two ICD-orders O and O^- for graphs G and $G \ominus \Delta G$ from the keynode and *cvs* vertices perspective. Specifically, we quantify their difference as $\overline{\texttt{diff}}(O, O^-) = \{\cup_{k=1}^{\delta^-} \overline{\texttt{diff}}_k\}$, where δ^- denotes the maximum core number of $G \ominus \Delta G$. Here, the $\overline{\texttt{diff}}_k$ is defined as:

$$\overline{\texttt{diff}}_k(O_k, O_k^-) = \left(\bigcup_{w \in \mathcal{D}_k^- \cap \mathcal{D}_k} \left\{ cvs_k[w] \cup cvs_k^-[w] \mid cvs_k[w] \neq cvs_k^-[w] \right\} \right) \cup \left(V_k \setminus V_k^- \right).$$

Note that typically $\overline{\texttt{diff}}_k \geq \texttt{diff}_k$, providing an additional tool for analyzing the time complexity of our algorithm. Unlike edge insertions, edge deletions require more computations.

Theorem 5.7. *The time complexity of* `OrdDel` *for deleting an edge (u, v) from a graph G is* $O\big(\sum_{k=1}^{\psi}(\mathrm{vol}(\overline{\texttt{diff}}_k) + |N(\overline{\texttt{diff}}_k, G_k)| \cdot \log |V_k|)\big)$.

Remark. We note that theoretically $\mathrm{vol}(|\texttt{diff}_k|) \leq \mathrm{vol}(|\overline{\texttt{diff}}_k|) \leq |E_k|$, and $|N(|\texttt{diff}_k|, G_k)| \leq |N(|\overline{\texttt{diff}}_k|, G_k)| \leq |V_k|$. Additionally, as shown in Table 5 and Table 6, we typically have $\mathrm{vol}(|\texttt{diff}_k|) < \mathrm{vol}(|\texttt{diff}_k|) \ll |E_k|$, and $|N(|\texttt{diff}_k|, G_k)| < |N(|\texttt{diff}_k|, G_k)| \ll |V_k|$ in real graphs.

Discussion. We note that our approach of maintaining ICs by propagating only updates on the ICD-order is conceptually similar to *Incremental View Maintenance (IVM)*, a fundamental technique for efficiently updating materialized views when base data changes [28, 29, 54]. In IVM, views are updated incrementally rather than recomputed from scratch, significantly reducing overhead. Theoretically, the tightest complexity bound of IVM is proportional to the size increase of the output [2], which is stricter than our current bound of $|\texttt{AFF}|$. In future work, we plan to further optimize our algorithm to approach this tighter bound and achieve lower time complexity.

6 Experiments

We now present the experimental results. Section 6.1 discusses the setup. We discuss the experimental results in Sections 6.2 and 6.3.

6.1 Setup

Datasets. We use eight real-world networks from different domains, which are downloaded from Stanford Network Analysis Platform [2], and Network Repository [3]. Table 4 provides the statistics of each graph, where n and m are the numbers of vertices and edges, respectively, and δ is the maximum core number. All algorithms are implemented in C++, compiled with the g++ compiler at -O3 optimization level, and run on a Linux machine with an Intel Xeon 2.40GHz CPU and 512GB RAM. If an algorithm cannot finish in 72 hours, we mark its running time as **INF**.

[2]http://snap.stanford.edu/data/

[3]https://networkrepository.com/network-data.php

Table 4. Dataset Statistics.

Dataset	Type	n	m	δ
Wikitalk (WK)	Communication	120,834	237,551	54
DBLP (DP)	Collaboration	317,080	1,049,866	118
web-Google (WG)	Web	5,105,040	4,322,052	44
as-skitter (AS)	Connection	1,694,616	11,094,209	111
Freebase (FB)	Hyperlink	14,420,276	40,655,068	17
Socfb-Uci-Uni (SU)	Social	58,790,783	92,208,195	16
UK-2002 (UK)	Web	18,483,186	261,787,258	942
IT-2004 (IT)	Web	41,291,594	1,027,474,947	3,224

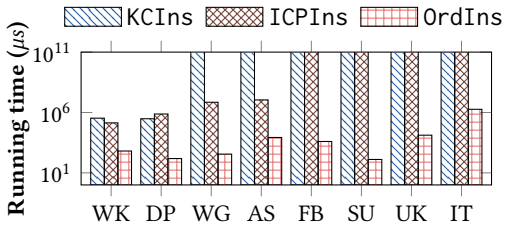


Fig. 7. Efficiency of an edge insertion on all datasets.

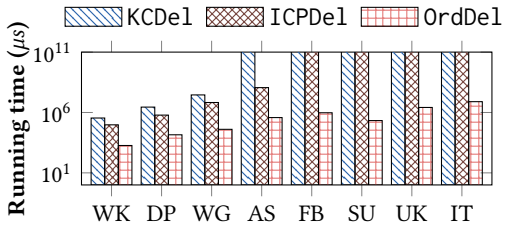


Fig. 8. Efficiency of an edge deletion on all datasets.

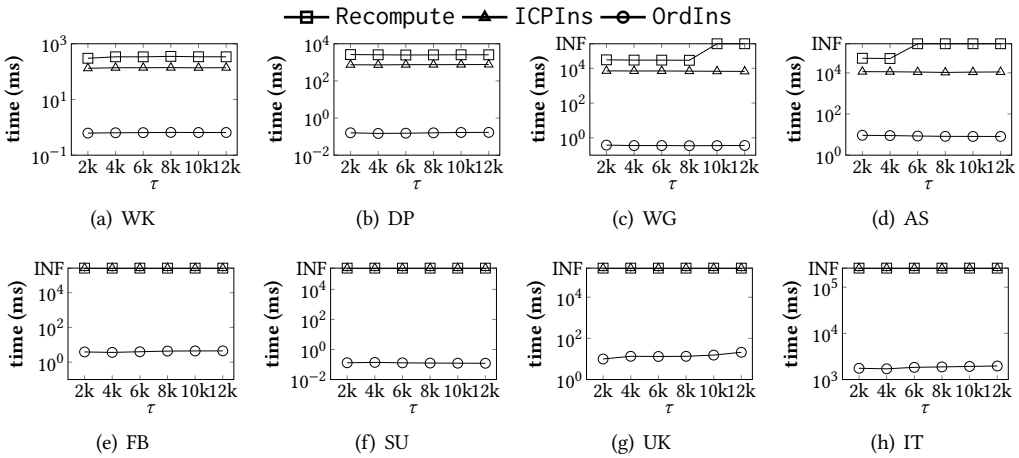
Algorithms. We compare the following algorithms:

- **KCIns:** the naive KC-Index maintenance algorithm under the edge insertion scenario (Section 3). It rebuilds the KC-Index for every edge insertion.
- **KCDel:** the naive KC-Index maintenance algorithm under the edge deletion scenario (Section 3). It rebuilds the KC-Index for every edge deletion.
- **ICPIns [42]:** the state-of-the-art ICP-Index maintenance algorithm under the edge insertion scenario.
- **ICPDel [42]:** the state-of-the-art ICP-Index maintenance algorithm under the edge deletion scenario.
- **OrdIns:** our proposed order-based KC-index maintenance algorithm under the edge insertion scenario.
- **OrdDel:** our proposed order-based KC-index maintenance algorithm under the edge deletion scenario.

Experimental settings. To evaluate the efficiency of influential community maintenance algorithms above, for each graph, we select τ edges to simulate the edge insertion and edge deletion process, where $\tau \in \{2K, 4K, 6K, 8K, 10K, 12K\}$. For the WK, DP, UK, and IT datasets, edges are inserted in chronological order (from oldest to newest) according to their timestamps; for the other datasets, edges are selected randomly. When testing edge insertions, we first remove the selected τ edges from the original graphs and then insert them back into the graph sequentially. When testing edge deletions, we directly delete these τ edges one by one from the original graphs. Note that the default value of τ is set to 6,000. In the following reported time cost, each data point is the average result of τ edge insertions or deletions.

6.2 Efficiency of Edge Insertions

We evaluate the performance of the edge insertion algorithms.



(a) WK (b) DP (c) WG (d) AS

(e) FB (f) SU (g) UK (h) IT

Fig. 9. Effect of the number of inserted edges (average time per edge).

Table 5. |AFF| after inserting 6,000 edges.

Datasets	Method		
	ICPIns	OrdIns	**Reduce**
WK	800,311,813	548,368	1,459 ×
DP	1,415,797,148	233,701	6,058 ×
WG	10,169,229,064	537,642	18,914 ×
AS	9,168,669,841	1,405,625	6,523 ×

Table 6. |AFF| after deleting 6,000 edges.

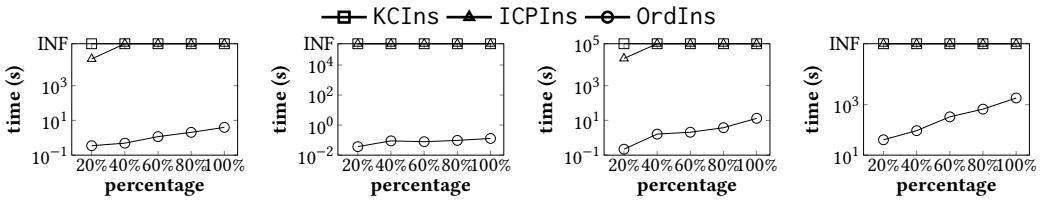
Datasets	Method		
	ICPDel	OrdDel	**Reduce**
WK	803,129,214	2,822,270	284 ×
DP	1,433,288,186	22,080,290	64 ×
WG	5,104,120,531	68,643,111	74 ×
AS	9,394,837,970	191,561,450	49 ×

▶ **Exp.1. Overall efficiency results.** Figure 7 compares the average running time of all algorithms on eight datasets by inserting 6K edges. Across all datasets, KCIns method exhibits the poorest performance. It cannot finish in 3 days on all the large datasets, as it redundantly recalculates all influential communities for all k in the graph, even for unmodified parts. Our algorithms show significant performance improvements compared to the baseline method. For example, on the SU dataset, OrdIns only needs 0.12ms to handle per edge insertion, while the SOTA algorithm, ICPIns takes more than 3 days. This indicates that our algorithm is up to six orders of magnitude faster than the SOTA algorithm. The primary reason for this improvement is that our proposed algorithm can accurately identify vertices impacted by edge insertions, allowing updates to be completed by accessing smaller subgraphs. In contrast, ICPIns needs to analyze these changes for every entire k-IC.

▶ **Exp.2. # of affected edges.** Table 5 depicts the total number of affected edges while inserting 6K edges. Our algorithm OrdIns demonstrates a notable up to 4 orders of magnitude reduction in the number of affected edges compared to the state-of-the-art algorithm ICPIns. Theoretically, the number of affected edges in OrdIns is $\sum_{k=1}^{\psi} \mathrm{vol}(\mathtt{diff_k})$, which is the dominator in its time complexity. This reduction aligns with our conclusion that only a few vertices need to change their positions in the new ICD-order, allowing OrdIns to check only a small subset of edges to correctly maintain the ICD-order. These results highlight that our ICD-order plays a crucial role in reducing the number of visited vertices, thus enhancing maintenance efficiency.

▶ **Exp.3. Effect of the number of inserted edges.** Figure 9 depicts the effect of the number of inserted edges on processing time, where $\tau \in \{2K, 4K, 6K, 8K, 10K, 12K\}$. We present the results on

all datasets. The results show that as the number of edges increases, the average insertion time of each edge does not increase significantly. This demonstrates the stability of our proposed algorithm across varying numbers of edge insertions.



(a) FB (b) SU (c) UK (d) IT

Fig. 10. The scalability test for handling edge insertions.

▸ **Exp.4. Scalability test.** To test the scalability, we randomly select 20%, 40%, 60%, 80%, and 100% of edges from each graph, and then obtain five induced subgraphs by these edges. We only show the results on the four largest datasets when inserting 6,000 edges in Figure 10 since the trends are similar on all other datasets. The running time of these three algorithms increases with the number of edges, our algorithm performs better than `Recompute` and `ICPIns` in all cases, and the growth rate of the curve of `OrdIns` is smaller.

▸ **Exp.5. Time proportion of different** k. Recall that `OrdIns` maintains the ICD-order for all k values from k=1 to δ. Figure 11 shows the time cost of `OrdIns` for processing different percentiles of k values, after inserting 6K edges (assuming the graph has been loaded into memory) on eight datasets. For most datasets, the time cost of updating the ICD-order for the first quartile of k is significantly higher than that for the remaining k values. This is because when k is small, $V(G_k)$ and $E(G_k)$ are relatively larger, so the number of vertices affected by inserting an edge is larger. Besides, for the last quartile of k, our algorithm typically takes less time.

6.3 Efficiency of Edge Deletions

In this section, we evaluate the performance of the edge deletion algorithms. We include the additional results on edge deletion in our supplementary material.

▸ **Exp.1. Overall efficiency results.** Figure 8 compares the average running time of these three algorithms on eight datasets by deleting 6K edges. Similar to edge insertions, `KCDel` method performs poorly on all datasets. However, unlike edge insertions, the performance gap between our algorithm and the strongest competitor (i.e., `ICPDel`) narrows, which aligns with our time complexity analysis above. Nevertheless, thanks to our ICD-order, `OrdDel` remains significantly

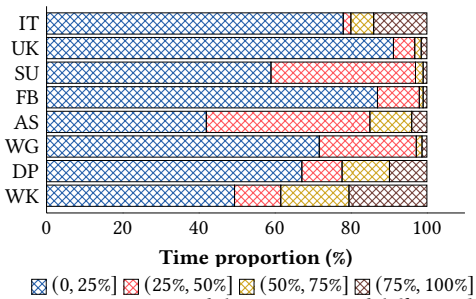


Fig. 11. Proportion of the time cost of different k in `OrdIns`.

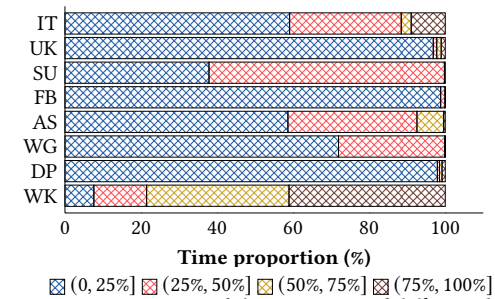


Fig. 12. Proportion of the time cost of different k in `OrdDel`.

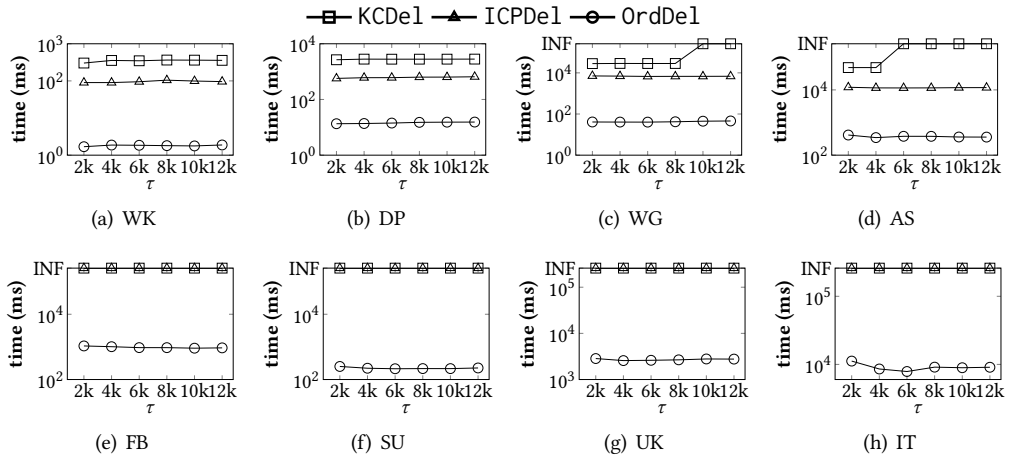


Fig. 13. Effect of the number of deleted edges (average time per edge).

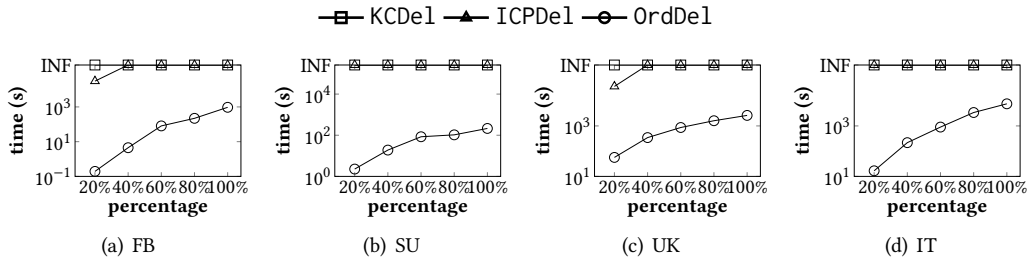


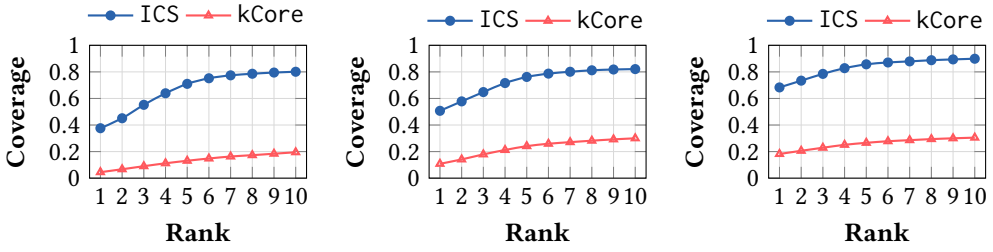
Fig. 14. The scalability test for handling edge deletions.

faster than ICPDel. For example, on the WG dataset, OrdDel only needs 226.12ms to handle per edge deletion, while the state-of-the-art algorithm, ICPDel takes more than three days. This indicates that our algorithm is up to three orders of magnitude faster than the state-of-the-art algorithm.

▶ **Exp.2. # of affected edges.** Table 6 depicts the total number of affected edges while deleting 6K edges. Our algorithm OrdDel demonstrates a notable up to 2 orders of magnitude reduction in the number of affected edges compared to ICPDel. Theoretically, the number of affected edges in OrdDel is $\sum_{k=1}^{\psi} \text{vol}(\overline{\texttt{diff}_k})$, which is the dominator in its time complexity. This result may explain why the performance gap between our algorithm and the strongest competitor narrows in deletion compared to insertion.

▶ **Exp.3. Effect of the number of deleted edges.** Figure 13 depicts the effect of the number of deleted edges on processing time, where $\tau \in \{2K, 4K, 6K, 8K, 10K, 12K\}$. We present the results on all datasets. Generally, the findings indicate that with the number of edges increasing, there is no substantial rise in the average time it takes to delete each edge. This demonstrates the stability of our proposed algorithm across varying numbers of edge deletions.

▶ **Exp.4. Scalability test.** To test the scalability, we randomly selected 20%, 40%, 60%, 80%, and 100% of edges from each graph, and then obtained five induced subgraphs by these edges. We show the results on all datasets when deleting 6,000 edges in Figure 14 since the trends are similar on all other datasets. The running time of these three algorithms increases with the number of edges, our algorithm performs better than Recompute and ICPDel in all cases, and the growth rate of the



(a) $k = 5$ (b) $k = 6$ (c) $k = 7$

Fig. 15. Comparison of the coverage ratio.

curve of OrdDel is smaller. Therefore, our proposed edge deletion algorithm scales well on large graphs in practice.

▶ **Exp.5. Time proportion of different** k. As OrdDel requires maintaining ICD-order for all $k \in [1, \delta]$, we analyze the computational overhead distribution through progressive k subdivisions. Figure 12 demonstrates the temporal heterogeneity when processing 6,000 edge deletions (post-memory initialization) across eight datasets. Notably, the algorithm exhibits distinct phase characteristics: The first quartile consumes nearly or over 60% of total computation time in six datasets. This phenomenon correlates with the scale of G_k subgraphs - smaller k values correspond to denser graph structures where edge modifications trigger cascading updates across broader vertex neighborhoods. Conversely, the last quartile demonstrates superior efficiency with time consumption below 10% in seven datasets. The contracted subgraph dimensions at higher k thresholds substantially reduce the propagation range of structural updates, resulting in localized computational adjustments.

6.4 Case study

In this section, we present a case study on ByteDance, one of the largest social network platforms. The case is based on a real-world dynamic graph, referred to as *ByteDanceData*, constructed from the company's monthly payment records. In *ByteDanceData*, each vertex represents a user and is assigned a weight indicating the user's cash-out risk. The company's goal is to identify users who have actually performed cash-out transactions from this social network. To achieve this, the company first locates groups of users with high cash-out risks and then manually verifies whether each of them actually performed cash-out transactions. To reduce manual effort, the company aims to extract the top-r ICs from the network and conduct manual verification only on these ICs, rather than on the entire graph.

To verify the effectiveness of our ICs, we conduct experiments on the ByteDanceData. Specifically, we compute the coverage ratio of the top-r ICs, where the coverage ratio is defined as the proportion of users who have performed cash-out transactions and are included within these ICs, relative to all such users in the dataset. Intuitively, a higher coverage ratio indicates that a larger portion of cash-out users are successfully identified. In our experiments, we extract the top-r ($r \in \{1, 2, \ldots, 10\}$) ICs for $k \in \{5, 6, 7\}$ on the *ByteDanceData* dataset. For comparison, we also compute the r k-cores and evaluate their coverage ratios under the same settings. As shown in Figure 15, our ICs consistently outperform the k-core baseline: for each value of k, the top-10 ICs cover over 80% of real cash-out users, whereas the naive k-cores cover only about 20%. These results demonstrate the effectiveness of ICs in identifying cash-out user groups and substantially reducing manual verification effort.

7 Related works

In this section, we review two representative groups of community retrieval methods: community detection (CD) and community search (CS) on both static and dynamic graphs.

Community detection (CD). In the literature, various CD methods have been proposed [25, 26, 53], such as spectral clustering [18, 71, 76], consensus clustering [27, 67], modularity-based approaches [13, 19, 52, 53], statistical inference-based approaches [36, 57], and structural similarity-based approaches [8, 10, 47, 58, 68, 70, 74, 78]. Some recent works [51, 60, 61, 63–66, 84] have studied CD on the other type of graphs. The maintenance of CD in dynamic graphs has also received much attention. Many structural similarity-based CD algorithms on dynamic graphs have been developed [58, 70, 78], which often maintain the index schema by updating the set of vertices whose similarity values need to be recomputed. Some works have studied conducting spectral clustering on the dynamic graphs by analyzing the change of approximate eigenvectors [17, 50, 55]. Recently, Laenen et al. [37] proposed the first algorithm for dynamic spectral clustering with a theoretical guarantee.

Community search (CS). CS aims to query densely connected subgraphs containing a specific vertex or a set of vertices [16, 23, 24, 62, 75, 82]. To measure the structure cohesiveness of a community, people often use the cohesive subgraph models [23], like k-core [1, 4], k-truss [14, 79], k-clique [15, 77, 83] and k-edge connected component [9, 30]. A representative group of CS works is based on the k-core model. For example, in [16, 62], the metric of minimum degree used in k-core is used for CS. Another group of CS works uses the k-truss [14, 79]. For example, in [31, 33], the k-truss-based model is used for CS. Besides, many CS works have considered vertices' attributes (e.g., [11, 22, 23, 32]). Particularly, to find the most influential communities, some works have considered vertices' importance values and studied the problem of influential CS [3, 12, 32, 39–42, 48, 49, 72, 81].

In addition, some researchers have studied the community search in dynamic graphs [21, 31, 43–45]. Many k-core maintenance algorithms have been developed [43, 59, 80], which often maintain k-cores by maintaining the core numbers of vertices. Li et al. [43] observed that when an edge is inserted or deleted, the set of vertices whose core numbers need to be updated is connected, and the set is in the sub-core where the edge is located, and proposed a quadratic-time algorithm. Sariyuce et al. [59] proposed an algorithm that is linear to the size of the subcore. Zhang et al. [80] proposed an order-based core number maintenance algorithm. Lin et al. [45] studied the k-core hierarchy maintenance problem.

8 Conclusions

In this study, we investigate the problem of influential community (IC) search over large dynamic graphs. We find that existing algorithms lack bounded guarantees and typically incur substantial redundant computations. To tackle this issue, we introduce a novel concept *ICD-order*, and utilize it to devise a relatively bounded IC maintenance algorithm for the edge insertion scenario. Additionally, we present an efficient algorithm for edge deletion with a more favorable time complexity. Our experimental evaluations on eight real-world graphs validate the efficacy of our proposed algorithms.

Acknowledgments

This work was supported in part by the 1+1+1 CUHK-CUHK(SZ)-GDSTC Joint Collaboration Fund under Grant 2025A0505000045, Guangdong Provincial Key Laboratory of Mathematical Foundations for Artificial Intelligence (2023B1212010001), Shenzhen Research Institute of Big Data under Grant SIF20240002, and ByteDance Collaboration Fund under Grant CT20240529123301.

References

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

[2] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. 2017. Answering Conjunctive Queries under Updates. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 303–318. doi:10.1145/3034786.3034789

[3] Fei Bi, Lijun Chang, Xuemin Lin, and Wenjie Zhang. 2018. An Optimal and Progressive Approach to Online Search of Top-K Influential Communities. *Proc. VLDB Endow.* 11, 9 (2018), 1056–1068.

[4] Francesco Bonchi, Arijit Khan, and Lorenzo Severini. 2019. Distance-generalized core decomposition. In *Proceedings of the 2019 International Conference on Management of Data*. 1006–1023.

[5] Lutz Bornmann and Hans-Dieter Daniel. 2007. What do we know about the h index? *Journal of the American Society for Information Science and technology* 58, 9 (2007), 1381–1385.

[6] Sergey Brin and Lawrence Page. 1998. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems* 30, 1-7 (1998), 107–117.

[7] Sylvain Brohee and Jacques Van Helden. 2006. Evaluation of clustering algorithms for protein-protein interaction networks. *BMC bioinformatics* 7 (2006), 1–19.

[8] Lijun Chang, Wei Li, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. pSCAN: Fast and Exact Structural Graph Clustering. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, Helsinki, Finland, 253–264. doi:10.1109/ICDE.2016.7498245

[9] Lijun Chang, Xuemin Lin, Lu Qin, Jeffrey Xu Yu, and Wenjie Zhang. 2015. Index-based optimal algorithms for computing steiner components with maximum connectivity. In *SIGMOD*. 459–474.

[10] Yulin Che, Shixuan Sun, and Qiong Luo. 2018. Parallelizing Pruning-based Graph Structural Clustering. In *Proceedings of the 47th International Conference on Parallel Processing (ICPP '18)*. Association for Computing Machinery, New York, NY, USA, 1–10. doi:10.1145/3225058.3225063

[11] Lu Chen, Chengfei Liu, Rui Zhou, Jianxin Li, Xiaochun Yang, and Bin Wang. 2018. Maximum co-located community search in large scale social networks. *Proceedings of the VLDB Endowment* 11, 10 (2018), 1233–1246.

[12] Shu Chen, Ran Wei, Diana Popova, and Alex Thomo. 2016. Efficient computation of importance based communities in web-scale networks using a single machine. In *CIKM*. 1553–1562.

[13] Aaron Clauset, M. E. J. Newman, and Cristopher Moore. 2004. Finding Community Structure in Very Large Networks. *Physical Review E* 70, 6 (Dec. 2004), 066111. arXiv:cond-mat/0408187 doi:10.1103/PhysRevE.70.066111

[14] Jonathan Cohen. 2008. Trusses: Cohesive subgraphs for social network analysis. *National security agency technical report* 16, 3.1 (2008).

[15] Wanyun Cui, Yanghua Xiao, Haixun Wang, Yiqi Lu, and Wei Wang. 2013. Online search of overlapping communities. In *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*. 277–288.

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

[17] Charanpal Dhanjal, Romaric Gaudel, and Stéphan Clémençon. 2014. Efficient eigen-updating for spectral graph clustering. *Neurocomputing* 131 (2014), 440–452.

[18] W. E. Donath and A. J. Hoffman. 1973. Lower Bounds for the Partitioning of Graphs. *IBM Journal of Research and Development* 17, 5 (Sept. 1973), 420–425. doi:10.1147/rd.175.0420

[19] J. Duch and A. Arenas. 2005. Community Detection in Complex Networks Using Extremal Optimization. *Physical Review E* 72, 2 (Aug. 2005), 027104. arXiv:cond-mat/0501368 doi:10.1103/PhysRevE.72.027104

[20] Wenfei Fan, Chunming Hu, and Chao Tian. 2017. Incremental graph computations: Doable and undoable. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 155–169.

[21] Yixiang Fang, Reynold Cheng, Yankai Chen, Siqiang Luo, and Jiafeng Hu. 2017. Effective and efficient attributed community search. *The VLDB Journal* 26 (2017), 803–828.

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

[23] Yixiang Fang, Xin Huang, Lu Qin, Ying Zhang, Wenjie Zhang, Reynold Cheng, and Xuemin Lin. 2020. A survey of community search over big graphs. *The VLDB Journal* 29, 1 (2020), 353–392.

[24] Yixiang Fang, Yixing Yang, Wenjie Zhang, Xuemin Lin, and Xin Cao. 2020. Effective and efficient community search over large heterogeneous information networks. *Proceedings of the VLDB Endowment* 13, 6 (2020), 854–867.

[25] Santo Fortunato. 2010. Community Detection in Graphs. *Physics Reports* 486, 3 (Feb. 2010), 75–174. doi:10.1016/j.physrep.2009.11.002

[26] Santo Fortunato and Darko Hric. 2016. Community Detection in Networks: A User Guide. *Physics Reports* 659 (Nov. 2016), 1–44. doi:10.1016/j.physrep.2016.09.002

[27] Andrey Goder and Vladimir Filkov. 2008. Consensus Clustering Algorithms: Comparison and Refinement. In *2008 Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*. Society for Industrial and Applied Mathematics, 109–117. doi:10.1137/1.9781611972887.11

[28] Timothy Griffin and Leonid Libkin. 1995. Incremental Maintenance of Views with Duplicates. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*. 328–339. doi:10.1145/223784.223849

[29] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. 1993. Maintaining Views Incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*. 157–166. doi:10.1145/170035.170066

[30] Jiafeng Hu, Xiaowei Wu, Reynold Cheng, Siqiang Luo, and Yixiang Fang. 2017. On minimal steiner maximum-connected subgraph queries. *IEEE Transactions on Knowledge and Data Engineering* 29, 11 (2017), 2455–2469.

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

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

[33] Xin Huang, Laks VS Lakshmanan, Jeffrey Xu Yu, and Hong Cheng. 2015. Approximate Closest Community Search in Networks. *Proceedings of the VLDB Endowment* 9, 4 (2015).

[34] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *International Conference on Learning Representations*.

[35] Yi-Xiu Kong, Gui-Yuan Shi, Rui-Jie Wu, and Yi-Cheng Zhang. 2019. k-core: Theories and applications. *Physics Reports* 832 (2019), 1–32.

[36] Johan H. Koskinen and Tom A.B. Snijders. 2007. Bayesian Inference for Dynamic Social Network Data. *Journal of Statistical Planning and Inference* 137, 12 (Dec. 2007), 3930–3938. doi:10.1016/j.jspi.2007.04.011

[37] Steinar Laenen and He Sun. 2024. Dynamic Spectral Clustering with Provable Approximation Guarantee. In *Forty-first International Conference on Machine Learning*.

[38] Vito Latora and Massimo Marchiori. 2007. A measure of centrality based on network efficiency. *New Journal of Physics* 9, 6 (2007), 188.

[39] Rong-Hua Li, Lu Qin, Fanghua Ye, Jeffrey Xu Yu, Xiaokui Xiao, Nong Xiao, and Zibin Zheng. 2018. Skyline Community Search in Multi-valued Networks. In *SIGMOD*. ACM, 457–472.

[40] Rong-Hua Li, Lu Qin, Jeffrey Xu Yu, and Rui Mao. 2015. Influential Community Search in Large Networks. *Proc. VLDB Endow.* 8, 5 (2015), 509–520.

[41] Rong-Hua Li, Lu Qin, Fanghua Ye, Guoren Wang, Jeffrey Xu Yu, Xiaokui Xiao, Nong Xiao, and Zibin Zheng. 2020. Finding skyline communities in multi-valued networks. *The VLDB Journal* 29, 6 (2020), 1407–1432.

[42] Rong-Hua Li, Lu Qin, Jeffrey Xu Yu, and Rui Mao. 2017. Finding influential communities in massive networks. *The VLDB Journal* 26, 6 (2017), 751–776.

[43] Rong-Hua Li, Jeffrey Xu Yu, and Rui Mao. 2013. Efficient core maintenance in large dynamic graphs. *IEEE transactions on knowledge and data engineering* 26, 10 (2013), 2453–2465.

[44] Xuankun Liao, Qing Liu, Xin Huang, and Jianliang Xu. 2024. Truss-based community search over streaming directed graphs. *Proceedings of the VLDB Endowment* 17, 8 (2024), 1816–1829.

[45] Zhe Lin, Fan Zhang, Xuemin Lin, Wenjie Zhang, and Zhihong Tian. 2021. Hierarchical core maintenance on large dynamic graphs. *Proceedings of the VLDB Endowment* 14, 5 (2021), 757–770.

[46] Boge Liu, Fan Zhang, Wenjie Zhang, Xuemin Lin, and Ying Zhang. 2021. Efficient community search with size constraint. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 97–108.

[47] Kaixin Liu, Sibo Wang, Yong Zhang, and Chunxiao Xing. 2023. An efficient algorithm for distance-based structural graph clustering. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–25.

[48] Wensheng Luo, Xu Zhou, Kenli Li, Yunjun Gao, and Keqin Li. 2021. Efficient influential community search in large uncertain graphs. *IEEE Transactions on Knowledge and Data Engineering* 35, 4 (2021), 3779–3793.

[49] Wensheng Luo, Xu Zhou, Jianye Yang, Peng Peng, Guoqing Xiao, and Yunjun Gao. 2020. Efficient approaches to top-r influential community search. *IEEE Internet of Things Journal* 8, 16 (2020), 12650–12657.

[50] Lionel Martin, Andreas Loukas, and Pierre Vandergheynst. 2018. Fast approximate spectral clustering for dynamic networks. In *International Conference on Machine Learning*. PMLR, 3423–3432.

[51] Lingkai Meng, Long Yuan, Zi Chen, Xuemin Lin, and Shiyu Yang. 2022. Index-Based Structural Clustering on Directed Graphs. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, Kuala Lumpur, Malaysia, 2831–2844. doi:10.1109/ICDE53745.2022.00257

[52] M. E. J. Newman. 2004. Fast Algorithm for Detecting Community Structure in Networks. *Physical Review E* 69, 6 (June 2004), 066133. arXiv:cond-mat/0309508 doi:10.1103/PhysRevE.69.066133

[53] M. E. J. Newman and M. Girvan. 2004. Finding and Evaluating Community Structure in Networks. *Physical Review E* 69, 2 (Feb. 2004), 026113. doi:10.1103/PhysRevE.69.026113

[54] Milos Nikolic and Dan Olteanu. 2018. Incremental View Maintenance with Triple Lock Factorization Benefits. In *Proceedings of the 2018 International Conference on Management of Data*. 365–380. doi:10.1145/3183713.3183758

[55] Huazhong Ning, Wei Xu, Yun Chi, Yihong Gong, and Thomas Huang. 2007. Incremental spectral clustering with application to monitoring of evolving blog communities. In *Proceedings of the 2007 SIAM International Conference on Data Mining*. SIAM, 261–272.

[56] You Peng, Song Bian, Rui Li, Sibo Wang, and Jeffrey Xu Yu. 2022. Finding Top-r Influential Communities under Aggregation Functions. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 1941–1954.

[57] Joerg Reichardt and Stefan Bornholdt. 2006. Statistical Mechanics of Community Detection. *Physical Review E* 74, 1 (July 2006), 016110. arXiv:cond-mat/0603718 doi:10.1103/PhysRevE.74.016110

[58] Boyu Ruan, Junhao Gan, Hao Wu, and Anthony Wirth. 2021. Dynamic Structural Clustering on Graphs. In *Proceedings of the 2021 International Conference on Management of Data*. ACM, Virtual Event China, 1491–1503. doi:10.1145/3448016.3452828

[59] Ahmet Erdem Sarıyüce, Buğra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V Çatalyürek. 2016. Incremental k-core decomposition: algorithms and evaluation. *The VLDB Journal* 25 (2016), 425–447.

[60] Chuan Shi, Yitong Li, Jiawei Zhang, Yizhou Sun, and S Yu Philip. 2016. A survey of heterogeneous information network analysis. *IEEE Transactions on Knowledge and Data Engineering* 29, 1 (2016), 17–37.

[61] Chuan Shi, Ran Wang, Yitong Li, Philip S Yu, and Bin Wu. 2014. Ranking-based clustering on general heterogeneous information networks by network projection. In *CIKM*. 699–708.

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

[63] Yizhou Sun, Charu C Aggarwal, and Jiawei Han. 2012. Relation strength-aware clustering of heterogeneous information networks with incomplete attributes. *arXiv preprint arXiv:1201.6563* (2012).

[64] Yizhou Sun, Jiawei Han, Peixiang Zhao, Zhijun Yin, Hong Cheng, and Tianyi Wu. 2009. Rankclus: integrating clustering with ranking for heterogeneous information network analysis. In *EDBT*. 565–576.

[65] Yizhou Sun, Brandon Norick, Jiawei Han, Xifeng Yan, Philip S Yu, and Xiao Yu. 2013. Pathselclus: Integrating meta-path selection with user-guided object clustering in heterogeneous information networks. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 7, 3 (2013), 1–23.

[66] Yizhou Sun, Yintao Yu, and Jiawei Han. 2009. Ranking-based clustering of heterogeneous information networks with star network schema. In *KDD*. 797–806.

[67] A. Topchy, A.K. Jain, and W. Punch. 2005. Clustering Ensembles: Models of Consensus and Weak Partitions. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 27, 12 (Dec. 2005), 1866–1881. doi:10.1109/TPAMI.2005.237

[68] Tom Tseng, Laxman Dhulipala, and Julian Shun. 2021. Parallel Index-Based Structural Graph Clustering and Its Approximation. In *Proceedings of the 2021 International Conference on Management of Data*. 1851–1864. arXiv:2012.11188 [cs] doi:10.1145/3448016.3457278

[69] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *International Conference on Learning Representations*.

[70] Dong Wen, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. 2019. Efficient Structural Graph Clustering: An Index-Based Approach. *The VLDB Journal* 28, 3 (June 2019), 377–399. doi:10.1007/s00778-019-00541-4

[71] Scott White and Padhraic Smyth. 2005. A Spectral Clustering Approach To Finding Communities in Graphs. In *Proceedings of the 2005 SIAM International Conference on Data Mining*. Society for Industrial and Applied Mathematics, 274–285. doi:10.1137/1.9781611972757.25

[72] Yanping Wu, Jun Zhao, Renjie Sun, Chen Chen, and Xiaoyang Wang. 2021. Efficient personalized influential community search in large networks. *Data Science and Engineering* 6, 3 (2021), 310–322.

[73] Xiaoliang Xu, Jun Liu, Yuxiang Wang, and Xiangyu Ke. 2022. Academic Expert Finding via (K, P)-Core based Embedding over Heterogeneous Graphs. In *ICDE*.

[74] Xiaowei Xu, Nurcan Yuruk, Zhidan Feng, and Thomas A. J. Schweiger. 2007. SCAN: A Structural Clustering Algorithm for Networks. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, San Jose California USA, 824–833. doi:10.1145/1281192.1281280

[75] Yixing Yang, Yixiang Fang, Xuemin Lin, and Wenjie Zhang. 2020. Effective and efficient truss computation over large heterogeneous information networks. In *ICDE*. IEEE, 901–912.

[76] Hao Yin, Austin R. Benson, Jure Leskovec, and David F. Gleich. 2017. Local Higher-Order Graph Clustering. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '17)*. Association for Computing Machinery, New York, NY, USA, 555–564. doi:10.1145/3097983.3098069

[77] Long Yuan, Lu Qin, Wenjie Zhang, Lijun Chang, and Jianye Yang. 2017. Index-based densest clique percolation community search in networks. *IEEE Transactions on Knowledge and Data Engineering* 30, 5 (2017), 922–935.

[78] Fangyuan Zhang and Sibo Wang. 2022. Effective Indexing for Dynamic Structural Graph Clustering. *Proceedings of the VLDB Endowment* 15, 11 (July 2022), 2908–2920. doi:10.14778/3551793.3551840

[79] Yikai Zhang and Jeffrey Xu Yu. 2019. Unboundedness and efficiency of truss maintenance in evolving graphs. In *SIGMOD*. 1024–1041.
[80] Yikai Zhang, Jeffrey Xu Yu, Ying Zhang, and Lu Qin. 2017. A fast order-based approach for core maintenance. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 337–348.
[81] Yingli Zhou, Yixiang Fang, Wensheng Luo, and Yunming Ye. 2023. Influential Community Search over Large Heterogeneous Information Networks. *Proceedings of the VLDB Endowment* 16, 8 (2023), 2047–2060.
[82] Yingli Zhou, Yixiang Fang, Chenhao Ma, Tianci Hou, and Xin Huang. 2024. Efficient Maximal Motif-Clique Enumeration over Large Heterogeneous Information Networks. *Proceedings of the VLDB Endowment* 17, 11 (2024), 2946–2959.
[83] Yingli Zhou, Qingshuo Guo, Yixiang Fang, and Chenhao Ma. 2024. A counting-based approach for efficient K-clique densest subgraph discovery. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–27.
[84] Yang Zhou and Ling Liu. 2013. Social influence based clustering of heterogeneous information networks. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. 338–346.

Received July 2025; revised October 2025; accepted November 2025