A Unified Framework for Dense Subgraph Maintenance over Dynamic Bipartite Graphs

ZITAN SUN, Hong Kong Baptist University, China
ZIHAN JIA, Hong Kong Baptist University, China
HONG CHENG, The Chinese University of Hong Kong, China
XIN HUANG, Hong Kong Baptist University, China
JEFFREY XU YU, The Hong Kong University of Science and Technology (Guangzhou), China

To identify dense subgraphs, many *specific-subgraph algorithms* are developed for decomposition and maintenance on dynamic graphs. However, it suffers from two significant limitations. First, for each kind of subgraphs, it needs the development of specific maintenance algorithms, which is *a costly effort*. This may also lead to *potentially missing solutions* for an existing subgraph (e.g., k-nucleus maintenance). Second, *no commonly useful properties and rules* can be derived from these quite different algorithms.

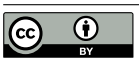
To address these bottlenecks, we propose a unified framework for dense subgraph maintenance over dynamic bipartite graphs. We first give a definition of (α, β)-core as bi-core and formulate the problem of bi-core maintenance. To our best knowledge, we are the first to propose the maintenance equivalence of bi-core to other subgraphs, e.g., k-core, k-truss, bi-truss, and k-nucleus. To handle the update efficiently, we propose a novel index structure of bi-core and onion layers, which finely decomposes one (α, β)-core into different levels of layers. This theoretically improves state-of-the-art bi-core maintenance algorithms from *unbounded* to *bounded*. However, due to two search dimensions of (α, β)-cores w.r.t. parameters α and β, it brings significant challenges for fast update. To tackle it, we develop a novel κBCO-index for fast maintenance by adding *a shortcut search of κ-direction*. This reorganizes bi-core onion layers and reduces the space complexity from $O(nd_{max})$ to $O(nk_{max})$, where n is the size of vertices, and the cardinality of κ-direction $k_{max} \ll d_{max}$ of the maximum degree holds in practice. Our proposed maintenance algorithms can handle a batch update of multiple edge insertions/deletions simultaneously. Extensive results on large datasets demonstrate that our unified framework efficiently maintains several dense subgraphs and runs faster than state-of-the-art bi-core maintenance algorithms.

CCS Concepts: • **Computing methodologies**; • **Theory of computation** → **Data structures and algorithms for data management**; • **Human-centered computing** → Collaborative and social computing; • **Networks** → Network algorithms;

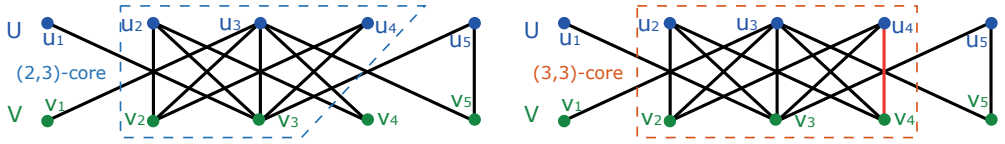
Additional Key Words and Phrases: bi-core, k-core, k-truss, nucleus, bitruss, incremental algorithms

ACM Reference Format:
Zitan Sun, Zihan Jia, Hong Cheng, Xin Huang, and Jeffrey Xu Yu. 2026. A Unified Framework for Dense Subgraph Maintenance over Dynamic Bipartite Graphs. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 4 (February 2026), 27 pages. https://doi.org/10.1145/3786618

Authors' Contact Information: Zitan Sun, zitansun@comp.hkbu.edu.hk, Hong Kong Baptist University, China; Zihan Jia, cszhjia@comp.hkbu.edu.hk, Hong Kong Baptist University, China; Hong Cheng, hcheng@se.cuhk.edu.hk, The Chinese University of Hong Kong, China; Xin Huang, xinhuang@comp.hkbu.edu.hk, Hong Kong Baptist University, China; Jeffrey Xu Yu, jeffreyxuyu@hkust-gz.edu.cn, The Hong Kong University of Science and Technology (Guangzhou), China.



ACM 2836-6573/2026/2-ART4
https://doi.org/10.1145/3786618



(a) The bipartite graph $G = (U, V, E)$ (b) The updated graph $G' = (U, V, E \cup \{(u_4, v_4)\})$

Fig. 1. A running example to show the (α, β)-core before and after inserting the edge (u_4, v_4).

1 Introduction

Graph is a fundamental model for representing relationships between entities, including social networks, traffic networks, chemical compounds, email networks, and so on. A dense subgraph that is formed by densely connected vertices, has many definitions, e.g., k-cliques [34], quasi-cliques [30], n-clans [27], n-club [27], k-core [1], k-truss [41], (α, β)-core (a.k.a. bicore) [2], bitruss [20, 42], and k-nucleus [33]. The cliques, quasi-cliques, n-clans, and n-club cannot be obtained in polynomial time. Fortunately, k-core, k-truss, (α, β)-core, k-bitruss, and k-nucleus for constant parameters can be obtained in polynomial time through a peeling process, which exhibits several desirable properties of high density and strong connectivity. Among numerous graph analytics tasks, dense subgraph search plays an important role, which has many useful practical applications such as community search [13–15, 21, 36, 45, 50], clique discovery [41], pivotal relationship identification [4, 35, 53], graph visualization [13, 50, 52], recommendation on customer-product bipartite graphs [24], and fraud user detection [42].

However, graphs could not be static but may dynamically update over time with node/edge insertions/deletions. Therefore, dense subgraphs also change in dynamic graphs, which seek to be maintained accordingly. In the literature, several recent works study *graph incremental algorithms* to maintain those dense subgraphs *individually one by one*, including k-core maintenance [1, 17, 50], k-truss maintenance [13, 36, 41, 49], and (α, β)-core maintenance [2, 24]. To our best knowledge, there is no existing work for maintaining two useful dense subgraphs of k-nucleus and bitruss yet. Even worse, although k-core, k-truss, (α, β)-core, bitruss, and k-nucleus share a similar peeling process for subgraph decomposition, there is unfortunately no unified framework for handling the general maintenance of these subgraphs with dynamic updates.

In light of the above, this work aims to propose a unified index for k-core, (α, β)-core, k-truss, k-nucleus, and bitruss, as well as a unified framework with general maintenance algorithms to handle dynamic updates. The key idea is to convert all these structures to the (α, β)-core, and then if we can efficiently maintain the (α, β)-core, we can also maintain other structures. We start with a basic concept of k-core. A k-core requires that each vertex has at least k neighbors. Fig. 1 (b) shows the subgraph within the dashed box as the 3-core, where all nodes have at least 3 neighbors. The bipartite graph $G = (U, V, E)$ is a kind of graph whose vertices can be divided into two groups U and V, and there is no edge within the same group of nodes. The (α, β)-*core* is an extension of k-core on bipartite graphs, where each vertex in U is connected to at least α vertices while each vertex in V is connected to at least β vertices. Fig. 1(a) shows an example of the $(2, 3)$-core for $\alpha = 2$ and $\beta = 3$, indicating that u_2, u_3, u_4 all have two neighbors and v_2, v_3 both have three neighbors. Consider the dynamic bipartite graph G has an update of inserting edge (u_4, v_4), the original $(2, 3)$-core has changed to a larger subgraph of $(3, 3)$-core as shown in Fig. 1(b). Therefore, we focus on studying the problem of bi-core maintenance by developing efficient maintenance techniques, which can be applied to accelerate the general maintenance framework and offer solutions for maintaining those subgraphs k-nucleus and bitruss.

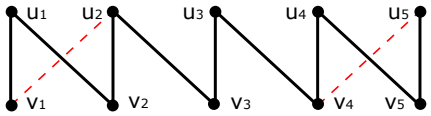


Fig. 2. An example shows that bi-core maintenance algorithm [24] is unbounded in case of inserting edges (v_1, u_2) or (v_4, u_5).

Limitations of existing bi-core maintenance. There is a state-of-the-art work of bi-core maintenance [24] based on an index of bi-core numbers. However, the usage of bi-core numbers has the following drawback. For example, in Fig. 2, ignoring all red dash lines, the bi-core numbers of v_3 is $\{(1, 2), (2, 1)\}$, indicating that v_3 is in the $(1, 2)$-core and the $(2, 1)$-core. The insertion of a single edge (u_2, v_1) or (u_5, v_4) cannot change the bi-core numbers of v_3. However, after inserting both edges (u_2, v_1) and (u_5, v_4), the whole graph is the $(2, 2)$-core, indicating that the insertion of either (u_2, v_1) or (u_5, v_4) actually affects the state of v_3 but the bi-core number cannot reflect this. Note that this kind of structure can be infinitely large and expanded by the following pattern, i.e., adding two nodes u_i and v_i and two edges (u_{i-1}, v_i) and (u_i, v_i), for $i > 1$. As we can see that, for correctly maintaining all bi-core numbers by [24], after inserting the edge (u_2, v_1), all nodes need to be visited, even almost all vertices have their bi-core numbers unchanged. Theoretically, given a graph G and the update ΔG, a good maintenance algorithm can finish updates in $O(P(||S||_h))$ time, where $P(.)$ is an arbitrary polynomial function, S represents nodes or edges that change their key states w.r.t. (α, β)-cores after the updates, and $||S||_h$ contains nodes and edges of h-hop neighbors of S. Therefore, we call this maintenance algorithm is a bounded algorithm and is bounded by S. However, the maintenance of bi-core number is *bounded in the case of edge deletion* but *unbounded in the case of edge insertion*. This indicates that, when inserting an edge into G, any maintenance algorithm cannot be bounded by nodes that change their bi-core numbers, and has to do extra works to visit unchanged nodes. In worst, these algorithms have to visit all nodes in G, leading to a huge cost of expensive computations.

To overcome this drawback, we propose a novel structure of bi-core onion layers, which decomposes (α, β)-core finely based on the peeling order. The onion layer contains all information of bi-core numbers. We maintain bi-core onion layers via the bounded algorithms, instead of maintaining bi-core numbers, for the efficiency. There are works using peeling order to maintain k-core [50] and k-truss [36, 49]. However, these techniques cannot be applied on (α, β)-core maintenance, because bi-core numbers are a two-dimensional index of α-direction and β-direction. Specifically, the (α, β)-core can be obtained from the $(\alpha - 1, \beta)$-core or from the $(\alpha, \beta - 1)$-core. A straightforward exploration of peeling process leads to combinatorial blow-ups with a total of $\binom{\alpha+\beta}{\alpha}$ different peeling orders. This brings significant technical challenges of bi-core maintenance. To address it, we propose a new κ-direction by select several main peeling orders and save them into our compact κBCO-index. The size of our κBCO-index is comparable to the graph size, which is space efficient for updating bi-cores. The contributions of this paper can be summarized as follows.

- We give the definition of (α, β)-core and analyze the properties of (α, β)-cores. We are the first to show the equivalence of (α, β)-core to other subgraphs of k-core, k-truss, k-bitruss, and k-nucleus by illustrating several detailed instances. Moreover, we formulate the bi-core maintenance problem and generalize it to a series of dense subgraph maintenance (Section 3).
- We propose the concept of bi-core gradient of (α, β)-cores w.r.t. parameter α and β. Based on it, we construct a novel BCO-index, which captures more information of bi-cores than bi-core numbers. We then propose bounded maintenance algorithms to update BCO-index for an edge insertion. Moreover, we analyze the algorithm complexity of BCO-index construction and BCO-index-based maintenance methods (Section 4).

- We design a new κ-direction for an effective peeling order. Based on it, we develop a compact κBCO-index to improve the BCO-index in terms of space complexity from $O(nd_{max})$ to $O(nk_{max})$, where the number of κ-directions as k_{max} is much smaller than the maximum degree d_{max}. We develop κBCO-index-based bounded maintenance algorithms to handle a batch of edge insertions/deletions. In addition, we extend to maintain nucleus over dynamic graphs (Section 5).
- We conduct extensive experiments on real large-scale bipartite graphs to show that our algorithms run much faster than the state-of-the-art methods [24]. The speedup can reach 8531x on dataset DBLP in the case of edge insertions. In addition, we conduct case studies of bi-core maintenance using five years' DBLP records and successfully maintain four other dense subgraphs for dynamic graphs (Section 6).

We discuss related works in Section 2 and conclude the paper in Section 7.

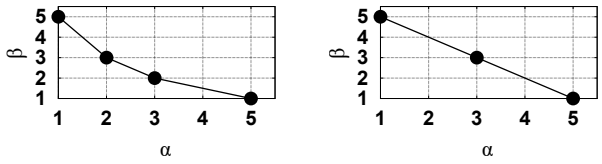
2 Related Work

Dense subgraph search. In the literature, there are numerous kinds of dense subgraph definitions, like k-clique [6, 34, 43], biclique [5, 26, 46], γ-quasi-clique [28, 32, 40], k-plex [7, 34], (k, l)-plex [11] and k-defective clique [3, 9, 47]. The problem of finding these dense subgraphs is NP-hard. In addition, a few dense subgraphs, e.g. k-core [1], (α, β)-core [2, 22], k-truss [8], k-(r, s)-nucleus [33] (constant parameters), and bitruss [42], obtained from the graph by peeling can be solved in polynomial time. The k-truss has been studied to enlarge by inserting new edges [35, 37]. All these methods are developed only for static graphs, which cannot handle the dense subgraph maintenance over dynamic graphs with the update of edge insertions/deletions. Other variants of dense subgraphs, like D-core and D-truss on directed graphs [12, 18, 19, 23] and uncertain graphs [38, 39, 44], that require a two-dimensional index can be inspired by our techniques.

Dynamic graph maintenance. When the graph dynamically updates, many works are devoted to quickly calculating the updated results, such as updating cliques [48], the k-nearest neighbors [25], the shortest path counting [10], butterflies counting [29], the k-centers [16], the top-t cores [51], and so on. Zhang et al. [49, 50] use order-based algorithms to maintain k-core and k-truss on dynamic graphs. Sun et al. [36] use the onion layer structure to maintain k-truss. These works cannot handle (α, β)-core maintenance, which cannot serve as the competitors against [24] and ours. As a result, the state-of-the-art method [24] is the only one competitor valid for the (α, β)-core maintenance task. Detailed comparison with [24] in terms of techniques and complexities can be found in Section 5.2.

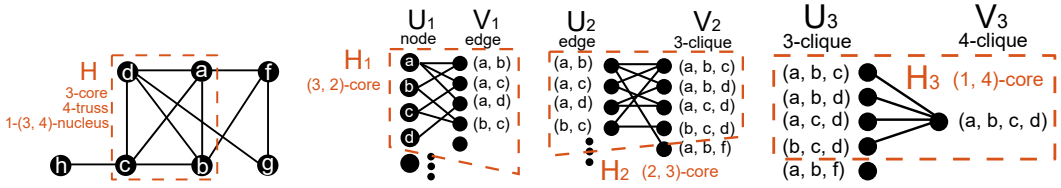
3 Preliminary

A general graph $\overline{G} = (\overline{V}, \overline{E})$ consists of a node set $\overline{V}$ and an edge set $\overline{E} \subseteq \overline{V} \times \overline{V}$. An induced subgraph $H = (\overline{V}(H), \overline{E}(H))$ of $\overline{G}$ contains a subset of nodes and all edges between them in $\overline{G}$, i.e., $\overline{V}(H) \subseteq \overline{V}$ and $\overline{E}(H) = \{(u, v) | u, v \in \overline{V}(H), (u, v) \in \overline{E}\}$. We use $d_H(p)$ to represent the degree of node p in H, i.e., $d_H(p) = |N_H(p)|$, where $N_H(p)$ is the set of neighbor nodes of p in H. We drop out the subscript $d_H(p)$ as $d(p)$ when the context is clear. A bipartite graph $G = (U, V, E)$ is an instance of general graphs $\overline{G}$, where U and V are two disjoint node sets and the edge set E only contains edges between U and V, i.e., $E \subseteq U \times V$. Note that there are no edges across two vertices in U, and neither V does. We use n and m to represent the numbers of nodes and edges in G, respectively, i.e., $n = |U| + |V|$ and $m = |E|$. We use $d_{\max}(U)$ and $d_{\max}(V)$ to represent the maximum degree of node in U and V, respectively. We use $d_{\max}$ to represent the maximum degree of node in G, i.e., $d_{\max} = \max_{p \in U \cup V} d(p)$.



(a) $\Phi(v_3)$ in original graph G (b) $\Phi'(v_3)$ in new graph G'

Fig. 3. The bi-core numbers of v_3 before and after insertion in Fig. 1. $\Phi(v_3) = \{(1,5),(2,3),(3,2),(5,1)\}$ and $\Phi'(v_3) = \{(1,5),(3,3),(5,1)\}$.



(a) General graph $\overline{G}$ (b) G_1 (c) G_2 (d) G_3

Fig. 4. An example of dense subgraph equivalence between the k-core, k-truss, k-(r,s)-nucleus H in general graph $\overline{G}$ to the bipartite (α,β)-cores H_1, H_2, H_3 in bipartite graphs G_1, G_2, and G_3, respectively.

3.1 Various Dense Subgraph Definitions

We present several dense subgraph definitions of k-core, (α,β)-core, k-truss, bitruss, and k-(r,s)-nucleus, as follows.

Definition 1 (k-core [1]). *Given a graph $\overline{G}$ and an integer $k \in \mathcal{Z}^{\geq 0}$, the subgraph $H \subseteq \overline{G}$ is the k-core* if and only if *H is the largest subgraph such that every vertex $p \in \overline{V}(H)$ has the degree $d_H(p) \geq k$.*

The core number $\theta(p)$. A useful concept for vertex p in k-core is the *core number*, denoted by $\theta(p)$, which is the largest value of $k \in \mathcal{Z}^{\geq 0}$ such that there exists a non-empty k-core $H \subseteq \overline{G}$ containing p. For example, consider the graph $\overline{G}$ in Fig. 1 (b), $\theta(v_3) = 3$, indicating that v_3 is in the 3-core, 2-core, and 1-core but not in any 4-core. Based on the definition of k-core in $\overline{G}$, we extend the definition of (α,β)-core in bipartite graph G.

Definition 2 ((α,β)-core [24]). *Given a bipartite graph $G = (U,V,E)$ and two integers $\alpha, \beta \in \mathcal{Z}^{\geq 0}$, the subgraph $H \subseteq G$ is the (α,β)-core* if and only if *H is the largest subgraph admits the degree constraint such that $d_H(u) \geq \alpha$ for $\forall u \in U$ and $d_H(v) \geq \beta$ for $\forall v \in V$ hold.*

***k*-truss and *k*-bitruss**. A triangle $\triangle_{vuw}$ is a small motif of 3-clique. The k-truss H of $\overline{G} = (\overline{V},\overline{E})$ [8] requires that every edge $e = (u,v) \in \overline{E}(H)$ is contained in at least $k-2$ triangles $\triangle_{vuw}$ for $w \in N(u) \cap N(v)$ within H. Similar to k-truss, the k-bitruss of $G = (U,V,E)$ [42] is defined based on a small motif of butterfly $\bowtie_{u_1u_2v_1v_2}$, where $u_1, u_2 \in U$, $v_1, v_2 \in V$, and $(u_i, v_j) \in E$ for $1 \leq i, j \leq 2$. Thus, the definition of k-bitruss is the largest subgraph $H \subseteq G$ such that every edge $e \in E(H)$ is contained in at least k butterflies in H.

Next, we present a generalized concept of k-nucleus based on k-clique, where a k-clique is a complete graph of k nodes such that every two nodes are connected.

Definition 3 (k-(r,s)-nucleus [33]). *Given a graph $\overline{G} = (\overline{V},\overline{E})$ and three integers k, r and s with $r < s$, the subgraph $H \subseteq \overline{G}$ is the k-(r,s)-nucleus if H is the union of all r-cliques that every r-clique is contained in at least k s-cliques.*

All the above dense subgraphs are cohesive controlled by a density parameter of k. The larger k, the denser graph H.

3.2 Properties of (α, β)-Core and Its Equivalence

Next, we introduce useful properties of (α, β)-cores.

Property 1 (Hierarchical Property). *Given two bi-cores (α_1, β_1)-core H_1 and (α_2, β_2)-core H_2, $H_1 \subseteq H_2$ holds if $\alpha_1 \geq \alpha_2$ and $\beta_1 \geq \beta_2$.*

Based on the hierarchical structure of (α, β)-cores, we define the bi-core numbers.

Definition 4 (Bi-Core Numbers Φ [24]). *Given a node p in bipartite graph G, the bi-core numbers of p are denoted by $\Phi(p) = \{(\alpha, \beta) : \alpha, \beta \in \mathcal{Z}^{\geq 0}, \exists$ a non-empty (α, β)-core $H \subseteq G$ s.t. $p \in V(H) \cup U(H)$, and $\nexists(\hat{\alpha}, \hat{\beta})$-core containing p with $(\hat{\alpha}, \hat{\beta}) > (\alpha, \beta)\}$.*

Note that the pair $(\hat{\alpha}, \hat{\beta}) > (\alpha, \beta)$ represents either $\hat{\alpha} > \alpha$, $\hat{\beta} \geq \beta$ or $\hat{\alpha} \geq \alpha$, $\hat{\beta} > \beta$. For example, $\Phi'(v_3) = \{(1, 5), (3, 3), (5, 1)\}$ in Fig. 3 (b), where $(3, 3)$ indicates v_3 is in the $(3, 3)$-core, also is in the $(2, 3)$-core or the $(3, 2)$-core and so on. Note that the bi-core numbers Φ by Definition 4 are difficult to quickly retrieve. To resolve it, in our work, we use a super set of bi-core numbers as $\Phi(.)$, where we add a few pairs of (α, β) qualified to meet Definition 2. The exact bi-core numbers can be easily obtained from $\Phi(.)$ by removing all (α, β) not meeting Definition 4.

For a node $p \in G$, $\Phi(p)$ is a set of a series of pairs of (α, β). We define $\alpha_{\max}(p)$ and $\beta_{\max}(p)$ as the maximum value of α and β among $\Phi(p)$, respectively. If we sort $\Phi(p)$ in the increasing order of α, we can get $\Phi(p) = \{(\alpha_1, \beta_{\max}(p)), \cdots, (\alpha_{\max}(p), \beta_1)\}$, where $\alpha_1, \cdots, \alpha_{\max}(p)$ and $\beta_1, \cdots, \beta_{\max}(p)$ are strictly increasing, respectively. Therefore, the number of pairs in $\Phi(p)$ is in $O(\min\{\alpha_{\max}(p), \beta_{\max}(p)\})$. For example, Fig. 3 (a) shows $\Phi(v_3)$ in Fig. 1 (a). It contains $(1, 5)$, $(2, 3)$, $(3, 2)$, $(5, 1)$. So $\alpha_{\max}(v_3) = 5$ and $\beta_{\max}(v_3) = 5$. If we treat the value of β in $\Phi(p)$ as an array and α as an array index and insert some pairs to make α continuous, the array is called $\beta_p = [\beta_{\max}(p), \cdots, \beta_1]$. The array β_p contains all information of $\Phi(p)$ and can quickly check whether the (α, β)-core contains p in $O(1)$ time. If a pair (α^*, β^*) is not in $\Phi(p)$, $\beta_p[\alpha^*]$ is equal to $\beta_p[\alpha^* + 1]$. For example, in Fig. 3 (a), $\beta_{v_3} = [5, 3, 2, 1, 1]$. The value of $\alpha = 4$ is not in $\Phi(v_3)$, so $\beta_{v_3}[4] = \beta_{v_3}[5] = 1$. The $(3, 3)$-core does not contain v_3, because $\beta_{v_3}[3] = 2 < 3$. The array α_p is defined similarly.

Equivalence to other subgraphs. We show the equivalence of (α, β)-cores to other subgraphs k-core, k-truss, k-(r, s)-nucleus, and k-bitruss, one by one as follows.

- First, (α, β)-core $\backsimeq$ k-core. Consider the k-core in a general graph $\overline{G} = (\overline{V}, \overline{E})$, we construct a corresponding bipartite graph $G_1 = (U_1, V_1, E_1)$. We assign two vertex sets $U_1 = \overline{V}$ and $V_1 = \{e = (u, v) \in \overline{E}\}$. For the edge set E_1, we connect $v, u \in U_1$ to $e = (u, v) \in V_1$ by adding two edges (v, e), (u, e) into E_1. Then, we set parameters $\alpha = k$ and $\beta = 2$. The (α, β)-core H_1 of G_1 corresponds to the subgraph $\overline{H}_1$in which every vertex has at least k valid edges within H_1, which is the k-core of $\overline{G}$. Fig. 4 (a) and (b) show an example of constructing G_1 based on $\overline{G}$. The subgraph H within the red dashed box is the 3-core of $\overline{G}$, while the subgraph H_1 in Fig. 4 (b) is the $(3, 2)$-core of G_1. The union of nodes in $U_1(H_1)$ is exactly H.
- Second, (α, β)-core $\backsimeq$ k-truss. Similar to the conversion of k-core, we can achieve it by assigning $G_2 = (U_2, V_2, E_2)$ where U_2 is the set of edges and V_2 is the set of triangles. Fig. 4(c) shows the $(2, 3)$-core H_2 of G_2 is the 4-truss of $\overline{G}$ in Fig. 4(a).
- Third, (α, β)-core $\backsimeq$ k-(r, s)-nucleus. We construct the $G_3 = (U_3, V_3, E_3)$, where $U_3 = \{$all r-cliques in $G\}$, $V_3 = \{$all s-cliques in $G\}$ and $E_3 = \{(u, v) : u \in U_3, v \in V_3, u \subseteq v\}$. As a result,

the $(k, \binom{s}{r})$-core of G_3 corresponds to the k-(r,s)-nucleus of $\overline{G}$. Similarly, in Fig. 4 (a), the subgraph H is the 1-$(3,4)$-nucleus of $\overline{G}$, while H_3 in Fig. 4 (d) is the $(1,4)$-core of G_3.

- Last but not least, (α,β)-core $\backsimeq$ k-bitruss. We construct the $G_4 = (U_4, V_4, E_4)$, where $U_4 = E$, $V_4 = \{\text{all butterflies } \bowtie \text{ in } G\}$ and $E_4 = \{(e_1 = (v_1, u_1), \bowtie_{v_1v_2u_1u_2}) : u_1, u_2 \in U, v_1, v_2 \in V, e_1 \in \bowtie_{v_1v_2u_1u_2}\}$. The $(k,4)$-core of G_4 corresponds to the k-bitruss of G.

3.3 Problem Formulation and Generalization

Based on the bi-core numbers by Definition 4, we can obtain the bi-core index of G as $\Phi(G) = \{\Phi(p) : \forall p \in U \cup V\}$. Therefore, we formulate the problem of bi-core maintenance as follows.

Problem 1 (Bi-core Maintenance). *Given a dynamic bipartite graph $G = (U, V, E)$ with an edge $e' = (u, v)$ insertion/deletion, and the original bi-core index $\Phi(G)$, the problem of bi-core maintenance is to recompute the* new bi-core index $\Phi'(G') = \{\Phi'(p) : \forall p \in U \cup V\}$, *where the new graph $G' = (U, V, E')$ has either $E' = E \cup \{e'\}$ for the edge insertion or $E' = E \setminus \{e'\}$ for the edge deletion.*

Throughout this paper, we use the symbol $'$ to represent the updated subgraph/concepts/index, e.g., G', $\Phi'(*)$. For example, in Fig. 1, G changes to G' after inserting (u_4, v_4). According to the equivalence of dense subgraphs in Section 3.2, we can generalize the problem of bi-core maintenance to handle alternative dense subgraph maintenance.

Problem 2 (General Maintenance). *Given a general graph $\overline{G}$, an update of graph region $\Delta\overline{G}$, two motif patterns A and B (e.g., nodes, edges, $\triangle$, $\bowtie$, $\boxtimes$), two integers α and β, an original index that keeps* an arbitrary dense subgraph A–B *where each A connects to at least α times of B and each B connects to at least β times of A, the problem of general maintenance is to update a new index on the new graph $\overline{G}' = \overline{G} \pm \Delta\overline{G}$, where the symbol $\pm$ indicates $+$ or $-$ for insertion or deletion, respectively.*

Our problem of general maintenance is a general version of bi-core maintenance by setting that A is the node in U and B is the node in V for a bipartite graph $G = (U, V, E)$. Moreover, the problem of general maintenance could be extended to handling other subgraph maintenance, leveraging the equivalence of dense subgraphs in Section 3.2. For example, consider a dynamic graph $\overline{G}$ for an edge insertion $e_0 = (u, v)$. For the k-core maintenance, we assign that A is the node and B is the edge, $\alpha = k$, $\beta = 2$, and add two edge insertions (u, e_0) and (v, e_0) in a bipartite graph G_1. Similarly, for the k-truss maintenance, we assign that A is the edge and B is the triangle, $\alpha = k-2$, $\beta = 3$, and add multiple edge insertions $(e_0, \triangle_{uvw_i})$, $((u, w_i), \triangle_{uvw_i})$, and $((v, w_i), \triangle_{uvw_i})$ for all $w_i \in N(u) \cap N(v)$ in a bipartite graph G_2. Similar conversions can be performed for k-nucleus, and k-bitruss. Note that although we focus on the bi-core maintenance, the proposed techniques can also be easily extended to tackle a series of variant general maintenance problems above. A successful instance of k-nucleus maintenance with detailed algorithms and results is reported in Section 5.3 and Exp-VII in Section 6, respectively.

4 Our Bi-core Onion Layer Index

In this section, we first present the bi-core onion layers and the gradient between two onion layers. Next, we present a novel index of bi-core onion layer called BCO-index. Finally, we develop algorithms for BCO-index construction and maintenance.

4.1 Bi-core Onion Layers and Gradient

The structure of bi-core onion layers. To find the (α,β)-core, we can adopt the peeling process to iteratively remove from G the nodes that do not meet the degree requirements, and obtain the result. Obviously, the bi-core number $\Phi(.)$ only keeps the final result and ignores the information

of the batch of node removal, which is the key to the update of bi-core. Therefore, for each vertex p, we keep both the information of $\Phi(p)$ and the removal batch of node p, which is defined as the *onion layer* for p.

Definition 5 (Onion Layer). *Given a vertex u where $u \in H$ and $u \notin H'$, the onion layer $L_{H\to H'}(u) \in \mathcal{Z}^{>0}$ is a positive integer that shows the batch number of the removal of u from H to H'.*

Note that in each batch, we remove all unqualified nodes together. If a node u exists in both H and H', we define $L_{H\to H'}(u) = \infty$. The peeling process is directional in our problem, so the onion layer is also directional. In the following, we give a new definition of *bi-core gradient* to represent onion layers in two orthogonal directions.

Definition 6 (Bi-core Gradient). *Given a node u and a bi-core number $P = (\alpha_1, \beta_1) \in \Phi(u)$, the bi-core gradient, $\nabla u(P) = (\frac{\partial u}{\partial \alpha}(P), \frac{\partial u}{\partial \beta}(P))$, of node u at point P is a pair of onion layers, which shows the batch of the removal of u in the computation of (α_1+1, β_1)-core and (α_1, β_1+1)-core, i.e., $\frac{\partial u}{\partial \alpha}(P) = L_{(\alpha_1,\beta_1)\text{-core}\to(\alpha_1+1,\beta_1)\text{-core}}(u)$ and $\frac{\partial u}{\partial \beta}(P) = L_{(\alpha_1,\beta_1)\text{-core}\to(\alpha_1,\beta_1+1)\text{-core}}(u)$.*

To simplify, we use $L^u_\alpha(P)$ to represent $\frac{\partial u}{\partial \alpha}(P)$. When the context is clear, we omit P or u or α, i.e., L^u_α or L^u or L_α. For example, consider the graph in Fig. 2, ignoring all red dash lines and the node v_1, the remained subgraph is the $H = (1,2)$-core. For obtaining the $H' = (2,2)$-core from the H, u_1 and u_5 are first removed, due to the degree less than 2. So $L^{u_1}_\alpha((1,2)) = L_{H\to H'}(u_1) = 1$ and $L_{H\to H'}(u_5) = 1$. After removing u_1 and u_5, the degree of v_2 and v_5 decreases, so $L_{H\to H'}(v_2) = 2$ and $L_{H\to H'}(v_5) = 2$. Repeating the above process, we get $L_{H\to H'}(u_2) = 3, L_{H\to H'}(u_4) = 3, L_{H\to H'}(v_3) = 4$, $L_{H\to H'}(v_4) = 4$ and $L_{H\to H'}(u_3) = 5$. For obtaining $H'' = (1,3)$-core from H, vertices v_2, v_3, v_4, v_5 are first removed, so $L_{H\to H''}(v_i) = 1$ for $2 \le i \le 5$. Then, u_1, u_2, u_3, u_4, u_5 are removed, so $L_{H\to H''}(u_i) = 2$ for $1 \le i \le 5$. Therefore, for node u_1 at point $(1,2)$, $\nabla u_1((1,2)) = (1,2)$.

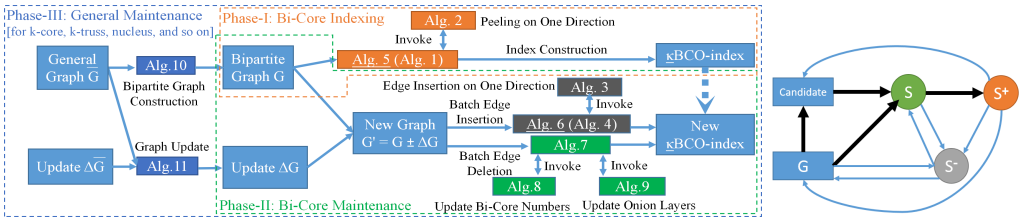
The structure of BCO-index. Based on the definition of bi-core number $\Phi(.)$ and bi-core gradient ∇, we design a novel structure of bi-core onion layer based index called BCO-index.

Definition 7 (BCO-index). *Given a node $u \in G$,* BCO-index(u) *consists of two parts: $\Phi(u)$ and ∇u, i.e.,* BCO-index$(u) = \{(P, \nabla u(P))\}, \forall P \in \Phi(u)$.

To capture all valid bi-core gradients, for a node u, we insert several pairs of (α, β) into $\Phi(u)$ that $\nabla u((\alpha,\beta)) \neq (\infty, \infty)$. The size of $\Phi(u)$ in Definition 4 is in $O(\min\{\alpha_{\max}(u), \beta_{\max}(u)\}$, while the size of $\Phi(u)$ in BCO-index is in $O(\alpha_{\max}(u) + \beta_{\max}(u))$. For example, Fig. 3 shows $\Phi(v_3) = \{(1,5), (2,3), (3,2), (5,1)\}$. For BCO-index, we should save the value of $\nabla v_3((\alpha,\beta))$ for all (α,β) in $[(1,5), (1,4), (2,3), (3,2), (4,1), (5,1)]$. The reason why we also keep $(1,4)$ is that v_3 is in $(1,4)$-core but not in $(2,4)$-core, so $L^{v_3}_\alpha$ is not ∞.

Consider the graph in Fig. 2, we give an example of how BCO-index solves the limitation of bi-core maintenance [24]. Ignoring all red dash lines, BCO-index$(v_3) = [((1,2), (4,1)), ((2,1), (2,5))]$. After inserting the edge (u_2, v_1) into G, BCO-index$'(v_3) = [((1,2), (6,1)), ((2,1), (2,5))]$. The insertion of (u_2, v_1) delays the removal of v_3 from $(1,2)$-core to $(2,2)$-core, indicating that BCO-index can accurately capture changes of the graph.

An overview of our unified maintenance framework. Fig. 5(a) shows an overview of unified maintenance framework. It mainly consists of three phases. Note that all algorithms will be detailed presented in later sections. First, in the Phase-I of bi-core indexing, we first design indexing algorithms (Algs. 1,5) to construct our BCO-index, which invokes decomposition procedure (Alg. 2) to peel G on one direction of α or β. Note that κBCO-index is an advanced and compact index of BCO-index. Next, in the Phase-II of bi-core indexing, we design two updating algorithms (Algs. 4,6,7) for batch edge insertion/deletions by invoking several procedures (Algs. 3,8,9), which



(a) A framework of our unified maintenance techniques (b) Node status conversion

Fig. 5. An overview of our unified maintenance framework and the node status conversion for maintaining a specific bi-core.

Algorithm 1 BCO-index Construction

Require: Graph $G = (U, V, E)$;
Ensure: BCO-index;
1: BCO-index $\leftarrow \emptyset$;
2: Apply Algorithm 2 with $(\underline{\alpha}, \underline{\beta}) = (1, 1)$ on α-direction and get $\alpha_{\max}$;
3: Apply Algorithm 2 with $(\underline{\alpha}, \underline{\beta}) = (1, 1)$ on α-direction and get $\beta_{\max}$;
4: **for** α from 2 to $\alpha_{\max}$ **do**
5: Apply Algorithm 2 with $(\underline{\alpha}, \underline{\beta}) = (\alpha, 1)$ on β-direction;
6: **for** β from 2 to $\beta_{\max}$ **do**
7: Apply Algorithm 2 with $(\underline{\alpha}, \underline{\beta}) = (1, \beta)$ on α-direction;
8: Add auxiliary points to BCO-index and sort BCO-index;
9: **return** BCO-index;

takes a specific direction for updating bi-core numbers and onion layers. Finally, in the Phase-III of general maintenance, we leverage the obtained bi-core maintenance techniques to design general updating algorithms for a series of dense subgraphs, e.g., k-core, k-truss, k-nucleus, and so on. This needs our graph construction algorithms (Algs. 10,11) to first convert geneal graphs to a bipartite graph and build the new bipartite update.

4.2 BCO-Index Construction

We present the algorithm to construct BCO-index for all vertices in Algorithm 1. The key idea is to compute the bi-core number of each vertex $v \in U \cup V$ and keep the last peeling round of v to be deleted as the onion layer. First, we fix $\beta = 1$ and peel G with the increase of α by using Algorithm 2 and get $\alpha_{\max}$ (line 2). Then, we fix $\alpha = 1$ and peel G with the increase of β to get $\beta_{\max}$ (line 3). Next, we fix α from 2 to $\alpha_{\max}$ and peel G with the increase of β. Similarly, we fix β from 2 to $\beta_{\max}$ and peel G with the increase of α (lines 4-6). Finally, we add some auxiliary points to make bi-core gradients for each node u in pairs and set empty values to ∞. In Algorithm 2, we peel G with the increase of α or β, depending on the direction. The variable l records the current onion layer by starting from $l = 1$. First, we remove all nodes not in the $(\underline{\alpha}, \underline{\beta})$-core (line 1), and initially set $\alpha = \underline{\alpha}$ and $\beta = \underline{\beta}$ (line 3). We increase α or β based on the direction (lines 6-8). Then, if there exists $u \in U$ with $d(u) < \alpha$ or $v \in V$ with $d(v) < \beta$, we put these nodes into N_R and remove them together (line 9). The value of $L^v_{direction}(P)$ is l and is saved into BCO-index (line 13). Next, we increase l by 1 (line 14) and repeat this process, until no node is removed. In the following, we give the complexity analysis of BCO-index.

Theorem 1. *The* BCO-index *construction in Algorithm 1 takes* $O(md_{\max})$ *time and* $O(nd_{\max})$ *space. The* BCO-index *size is* $O(nd_{\max})$.

Algorithm 2 Peeling on One Direction

Require: Graph $G = (U, V, E)$, BCO-index, $\underline{\alpha}$, $\underline{\beta}$, $direction$;
Ensure: BCO-index;
1: **while** $\exists p \in U$ with $d(p) < \underline{\alpha}$ or $\exists p \in V$ with $d(p) < \underline{\beta}$ **do**
2: Remove p from G and update the degree of $p's$ neighbors;
3: Initialize $\alpha = \underline{\alpha}$ and $\beta = \underline{\beta}$;
4: **while** $U \neq \emptyset$ or $V \neq \emptyset$ **do**
5: Set $l \leftarrow 1$, $P \leftarrow (\alpha, \beta)$;
6: **if** $direction =$ "α" **then** Set $\alpha \leftarrow \alpha + 1$;
7: **if** $direction =$ "β" **then** Set $\beta \leftarrow \beta + 1$;
8: **if** $direction =$ "κ" **then** Set $\alpha \leftarrow \alpha + 1$ and $\beta \leftarrow \beta + 1$;
9: **while** $\exists u \in U$ with $d(u) < \alpha$ or $\exists v \in V$ with $d(v) < \beta$ **do**
10: $N_R \leftarrow \{u \in U | d(u) < \alpha\} \cup \{v \in V | d(v) < \beta\}$;
11: Remove N_R from G and update the degree of neighbors;
12: **for all** $v \in N_R$ **do**
13: Add $L^v_{direction}(P) = l$ to BCO-index;
14: $l \leftarrow l + 1$;
15: **return** BCO-index;

Proof. Each peeling process of Algorithm 1 takes $O(m)$ time. The time complexity of Algorithm 1 is $O(m(\alpha_{\max} + \beta_{\max})) = O(m \cdot 2d_{\max}) = O(md_{\max})$. For each node $p \in U \cup V$, the size of BCO-index(p) is in $O(\alpha_{\max}(p) + \beta_{\max}(p))$. Therefore, BCO-index takes $O(\sum_{p \in U \cup V} (\alpha_{\max}(p) + \beta_{\max}(p))) = O(nd_{\max})$ space. Algorithm 1 stores the whole BCO-index in $O(nd_{\max}))$ space. □

4.3 BCO-Index Maintenance

First, we give some properties of bi-core gradient to help analyze the increase of bi-core numbers and onion layers of nodes. Then, we introduce rules for changing statuses of nodes, which can guarantee correct maintenance. Finally, we propose our maintenance algorithms as well as the theoretical analysis.

Definition 8 (Onion Layer Subgraph $\mathcal{H}$). *Given a graph G, a bi-core number $P = (\alpha_1, \beta_1)$, an onion layer l, the onion layer subgraph $\mathcal{H}^l_\alpha(P)$ on α direction is an induced subgraph of G, where every node p in it is in the (α_1, β_1)-core and $L^p_\alpha \geq l$. Similarly, every node p in $\mathcal{H}^l_\beta(P)$ is in the (α_1, β_1)-core and $L^p_\beta \geq l$.*

Given the definition, we can conclude that $\mathcal{H}^{l+1}_\alpha(P) \subseteq \mathcal{H}^l_\alpha(P)$. When the context is clear, we omit P or α or l, i.e., $\mathcal{H}^l_\alpha$ or $\mathcal{H}^l$ or $\mathcal{H}$. If a node p has $L^p_\alpha = l$, $\mathcal{H}^l$ is exactly the subgraph where p is to be removed in the peeling process.

Lemma 1 (Non-decreasing Bi-core Gradient for Insertion). *For all nodes p in G, after inserting an edge into G, we have values of L^p non-decreasing, which indicates $L'^p \geq L^p$ after insertion.*

Proof. As new edges are inserted into G, $d'(p) \geq d(p)$, p is to be removed later. Thus, p has a larger onion layer in G'. □

Due to Lemma 1, to reduce examined pairs of Φ, we update each bi-core gradient in the increasing order of α and β.

Definition 9 (Onion Layer Degree $\mathcal{D}$). *Given a node p, a pair (α_1, β_1) and an onion layer l, we define $\mathcal{D}^l_\alpha(p, (\alpha_1, \beta_1)) = d_{\mathcal{H}^l_\alpha((\alpha_1, \beta_1))}(p)$.*

The definition of $\mathcal{D}^l_\beta(p,(\alpha_1,\beta_1))$ on β-direction is similar. When the context is clear, we omit (α_1,β_1) or α or l, i.e., $\mathcal{D}(p)$. Similar to $\mathcal{H}$, $\mathcal{D}(p)$ is exactly the degree of the subgraph where p is to be removed in the peeling process. Through observation, we have discovered the following degree property. It indicates that the onion layers needs to be updated if the property violated during the update process.

Property 2 (Degree Property). *Given a node $u \in U$ that is contained in (α,β)-core but is not contained in $(\alpha{+}1,\beta)$-core, we have $\mathcal{D}_\alpha^{L^u_\alpha}(u,(\alpha,\beta)) = \alpha$ if $L^u_\alpha = 1$. If $L^u_\alpha > 1$, we have $\mathcal{D}_\alpha^{L^u_\alpha}(u,(\alpha,\beta)) \le \alpha$ and $\mathcal{D}_\alpha^{L^u_\alpha-1}(u,(\alpha,\beta)) > \alpha$. Similarly, given a node $u \in U$ that is contained in (α,β)-core but is not contained in $(\alpha,\beta{+}1)$-core, we have $L^u_\beta > 1$. We also have $\mathcal{D}_\beta^{L^u_\beta}(u,(\alpha,\beta)) < \alpha$ and $\mathcal{D}_\beta^{L^u_\beta-1}(u,(\alpha,\beta)) \ge \alpha$.*

For nodes $v \in V$, the degree property is similar as $u \in U$. For example, consider the graph in Fig. 2 without all red dash lines, since $L_\alpha^{u_5}((1,2)) = 1$, $\mathcal{D}^1_\alpha(u_5,(1,2))$ must be 1. At the same time, $L_\alpha^{u_4}((1,2)) = 3$, so $\mathcal{D}^3_\alpha(u_4,(1,2)) = 1 \le 1$ and $\mathcal{D}^2_\alpha(u_4,(1,2)) = 2 > 1$. Also, $L_\beta^{u_5}((1,2)) = 2 > 1$, thus $\mathcal{D}^2_\beta(u_5,(1,2)) = 0 < 1$ and $\mathcal{D}^1_\beta(u_5,(1,2)) = 1 \ge 1$. Importantly, we use $\epsilon_{\mathcal{H}}(p)$ to indicate whether a node p satisfies the degree property in $\mathcal{H}$ or not. When the context is clear, we omit $\mathcal{H}$, i.e., $\epsilon(p)$. For edge insertion, $\epsilon(p) = false$, indicating the increase of onion layers. For example, after inserting the edge (u_5,v_4) in Fig. 2, $\mathcal{D}^1_\alpha(u_5,(1,2)) = 2 > 1$, the onion layer of u_5 should be updated. Similarly, when deleting edges, $\epsilon(p) = false$, indicating the decrease of onion layers. For example, if we remove the edge (u_4,v_5) in Fig. 2, $\mathcal{D}^2_\alpha(u_4,(1,2)) = 1 \le 1$, u_4 should be removed in lower layer.

Updating rules for changing node statuses. Based on the degree property, we can easily find nodes needed to be updated. Note that the increase of degree of nodes cannot guarantee the increase of bi-core numbers. So, the key idea for insertion maintenance is to assume nodes not meeting Property 2 all can increase bi-core numbers and then draw back nodes that cannot increase bi-core numbers. Thus, we need four node sets, *Candidate*, S, S^+, S^-, and their relationships are shown in Fig. 5 (b). *Candidate* contains all nodes not meeting Property 2, together with affected pairs in Φ. For a specific pair (α,β), S keeps nodes that may increase (α,β) in BFS manner. S^+ keeps nodes handled by S. S^- contains nodes that cannot increase (α,β) but participated in degree calculation of nodes in S and S^+.

Algorithm 3 shows how to process S, S^+ and S^-. Given G, BCO-index, for a specific α, β and peeling direction, S is a seed set not meeting Property 2. For nodes p in S, we will temporarily set their onion layers as ∞, move them into set S^+ (line 11). We also keep the neighbor q into S if the degree violates Property 2. (line 14). Otherwise, q cannot be in the new (α',β')-core but involves in the degree of p, so we keep q into the set S^- (line 16), which will be removed in lines 4-10. The removal of nodes in S^- will decrease the degree of nodes in S^+. Therefore, when the degree of nodes in S^+ meets Property 2 again, we remove them from S^+ and update their onion layers as $L^q + 1$ (line 8).

For example, after inserting (u_2,v_1) in Fig. 2, we use Algorithm 3 to maintain onion layers from $(1,2)$-core to $(2,2)$-core. Suppose we have processed bi-core number $(1,1)$ and get $L_\alpha^{v_1}((1,2)) = 1$. Since v_1's neighbor u_1 has $\mathcal{D}^1(u_1) = 2 > 1$, so $S = \{u_1\}$. First, we pop u_1 and get $l = L^{u_1} = 1$ (line 3). Currently, $S^- = \emptyset$, we directly move to line 11 and set $L^{u_1} = \infty$ and push u_1 into S^+. We visit neighbors of u_5 and find $\mathcal{D}^1(v_1) = 2 = \mathcal{D}^2(v_2) \ge 2$, so $S = \{v_1,v_2\}$. Then, we pop v_1 because $L^{v_1} = 1$ is the smallest onion layer of nodes in S. After handled v_1, v_2 and u_2, we have $S^+ = \{u_1,u_2,v_1,v_2\}$ and $S = \{v_3\}$. We pop v_3 from S and set $L^{v_3} = \infty$ and push v_3 into S^+. There is only one neighbor u_3 of v_3 that is not in $S \cup S^+$. But $\mathcal{D}^5(u_3) = 1 \le 1$, so u_3 cannot be put into S and $S^- = \{u_3\}$. Since $S = \emptyset$, we pop all nodes in S^- (line 17). We pop u_3 in S^- and update $\mathcal{D}^\infty(v_3) = 1 < 2$ and set

Algorithm 3 Maintenance in One Direction for Edge Insertion

Require: Graph $G = (U, V, E)$, BCO-index, α, β, $direction$, S;
Ensure: BCO-index, S^+;
1: $S^+ \leftarrow \emptyset$, $S^- \leftarrow \emptyset$;
2: **while** $S \neq \emptyset$ **do**
3: Pop a node p in S with the smallest L^p and set $l \leftarrow L^p$;
4: **while** $\exists q \in S^-$ with $L^q < l$ **do**
5: Pop q from S^-;
6: **for all** $t \in N(q)$ **do**
7: **if** $t \in S^+$ and $\mathcal{D}(t) \leq \alpha$ **then**
8: Set $L^t \leftarrow L^q + 1$; Move t from S^+ to S^-;
9: **if** $t \in S$ **then**
10: Remove t from S and push t into S^-;
11: Set $L^p \leftarrow \infty$ and push p to S^+;
12: **for all** $q \in N(p)$ and $q \notin S \cup S^+$ **do**
13: **if** $L^q > l$ and $\mathcal{D}(q) > \alpha$ **then**
14: Push q into S; Remove q from S^-;
15: **else if** $L^q \geq l$ **then**
16: Push q into S^-;
17: Set $l \leftarrow \infty$ and conduct lines 4-10;
18: **return** BCO-index, S^+;

Algorithm 4 BCO-index Maintenance for Edge Insertion

Require: Graph $G = (U, V, E)$, BCO-index, an edge (u, v) to be inserted;
Ensure: BCO-index;
1: $Candidate_\alpha$, $Candidate_\beta$, $S \leftarrow \emptyset$; Add (u, v) to G;
2: **for all** $((\alpha, \beta), (L^u_\alpha, L^u_\beta)) \in$ BCO-index(u) **do**
3: **if** $\epsilon(u) = false$ **then**
4: Push $(\alpha, \beta, L^u_\alpha, u)$ into $Candidate_\alpha$;
5: Similar operations for β-direction;
6: Similar operations for node v, like lines 2-5;
7: **while** $Candidate_\alpha \neq \emptyset$ **do**
8: Find the smallest $\underline{\alpha} + \underline{\beta}$ in $Candidate_\alpha$;
9: Remove all records with the same $\underline{\alpha}$ and $\underline{\beta}$, $L_\alpha \neq 0$ in $Candidate_\alpha$ and keep these nodes to S;
10: Call Algorithm 3 by α-direction and get BCO-index and S^+;
11: Set $L^p_\alpha((\underline{\alpha} + 1, \underline{\beta})) \leftarrow 1$ in BCO-index(p), $\forall p \in S^+$;
12: **for all** $p \in S^+ \cup N(S^+)$ **do**
13: **if** $\epsilon(p) = false$ **then**
14: Push $(\underline{\alpha} + 1, \underline{\beta}, 1, p)$ into $Candidate_\alpha$;
15: Similar operations for $Candidate_\beta$, like lines 7-14;
16: **return** BCO-index;

$L^{v_3} = L^{u_3} + 1 = 6$ and move v_3 from S^+ to S^-. Finally, we return BCO-index and $S^+ = \{u_1, u_2, v_1, v_2\}$, where nodes increase their bi-core number from $(1, 2)$ to $(2, 2)$.

Maintain BCO-index for single edge insertion. Algorithm 4 shows the concrete steps. Since neighbors of nodes u and v have changed, we check every onion layer degree of u and v in BCO-index (lines 2-6); If the degree violates the Property 2, it indicates the onion layer should increase. We keep these nodes and the corresponding (α, β) into the sets $Candidate_\alpha$ and $Candidate_\beta$. We handle

Table 1. An example of κBCO-index(u_2) for node u_2 in Fig. 2.

Node	Neighbors	κ	α	β
u_2	v_2, v_3	$(1, 4)$		
u_2'	v_1, v_2, v_3	$(2, 1)$	$[(2, 2)]$	$[(2, 1)]$

each record with the increase order of $\alpha + \beta$ (lines 8-15). For a specific $\underline{\alpha}$ and $\underline{\beta}$ in α-direction, we first find all matched nodes in $Candidate_\alpha$ and keep them into S (line 9). Then, we conduct Algorithm 3 to maintain BCO-index and get S^+ where nodes are in the $(\underline{\alpha}+1, \underline{\beta})$-core. We add new index $L_\alpha^p((\underline{\alpha}+1, \underline{\beta})) = 1$ into BCO-index (line 11). Again, if the degree of nodes in the new $(\underline{\alpha}+1, \underline{\beta})$-core violates the Property 2, we save them into $Candidate_\alpha$ (lines 12-14). The operation for handling $Candidate_\beta$ is similar with handling $Candidate_\alpha$ (line 15). The algorithm terminates by updating BCO-index after checking all candidates.

For example, after inserting the edge (u_2, v_1) in Fig. 2, Algorithm 4 first checks all bi-core numbers of u_2 and v_1 (lines 2-6). So we get $Candidate_\alpha = \{(2, 1, 1, u_2)\}$ and $Candidate_\beta = \{(1, 1, 1, v_1), (2, 1, 1, v_1)\}$. We first handle $Candidate_\alpha$ (line 7) and get $direction = \alpha$, $\underline{\alpha} = 2$, $\underline{\beta} = 1$ and $S = \{u_2\}$ (line 9). Then we use Algorithm 3 and get $S^+ = \{u_2, v_1, v_2, v_3\}$. We set nodes in S^+ with bi-core number $(3, 1)$ and onion layer $L_\alpha = 1$ (line 11). Next, we check 1-hop neighborhood of nodes in S^+ and get $Candidate_\alpha = \{(3, 1, 1, v_1), (3, 1, 1, v_2), (3, 1, 1, v_3)\}$ and $Candidate_\beta = \{(1, 1, 1, v_1), (2, 1, 1, v_1), (3, 1, 1, u_2)\}$. Repeating above processes until all candidates are handled, our bi-core maintenance ends.

Theorem 2. *Algorithm 4 is bounded by* L_CHG *and takes* $O(||\text{ L\_CHG }||_1)$ *time and uses* $O(nd_{\max})$ *space, where* L_CHG *is the changed part of* BCO-index *and* $||\text{L\_CHG}||_1$ *denotes the size of the 1-hop neighborhood of* L_CHG.

Proof. First of all, we prove that Algorithm 4 is bounded by L_CHG. Following [31], the maintenance algorithm on a graph is bounded if the algorithm will only visit the h-hop neighborhood of nodes whose state variables will be modified after maintenance. Obviously, only state of nodes in L_CHG changed after applying our Algorithm 4. So next, we will show that Algorithm 4 will only visit the 1-hop neighborhood of L_CHG. Before handling nodes p with onion layer l in S in Algorithm 3, nodes q in S^- with $L^q < l$ are removed. If p is still in S, p must increase its onion layer and is moved into S^+. Therefore, nodes in S^+ are in L_CHG, and nodes used to be in S and S^- are in the 1-hop neighborhood of L_CHG. The maximum size of S and S^- is $||\text{L\_CHG}||_1$, so the time complexity of Algorithm 4 is $O(||\text{L\_CHG}||_1)$. Then, Algorithm 4 keeps BCO-index for all nodes in the graph, which takes $O(nd_{\max})$ space. So Algorithm 4 uses $O(nd_{\max})$ space. □

5 An Improved κBCO-Index

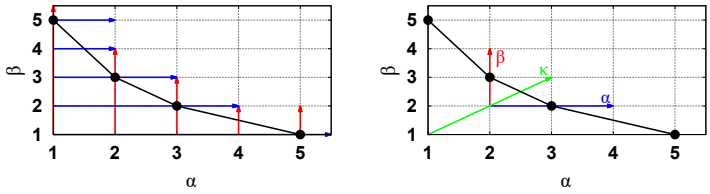
In this section, we propose a compact structure of κBCO-index, which improves BCO-index using smaller space. Based on κBCO-index, we develop maintenance algorithms for batch insertion/deletion, and extend to handle general maintenance for k-nucleus.

5.1 Compact κBCO-Index

Motivation. For any edge $(u, v) \in E$, u must be in the $(d(u), 1)$-core and v must be in the $(1, d(v))$-core. Therefore, $\alpha_{\max} = d_{\max}(U)$ and $\beta_{\max} = d_{\max}(V)$. In real-world datasets, $d_{\max}(U)$ and $d_{\max}(V)$ may be a very large number, leading to a huge space cost and time cost of BCO-index. Therefore, we can improve our index using a smaller space. First of all, as we mentioned before, any edge $(u, v) \in E$ is in the $(d(u), 1)$-core and the $(1, d(v))$-core. Thus, it is not necessary to keep the index for $\alpha = 1$ or $\beta = 1$, whose size equals to the size of the graph. Secondly, the (α, β)-core can be

Algorithm 5 κBCO-index Construction

Require: Graph $G = (U, V, E)$;
Ensure: κBCO-index;
1: κBCO-index $\leftarrow \emptyset$;
2: Apply Algorithm 2 with $(\underline{\alpha}, \underline{\beta}) = (1, 1)$ on κ-direction and get $k_{\max}$;
3: **for** k from 2 to $k_{\max}$ **do**
4: Apply Algorithm 2 with $(\underline{\alpha}, \underline{\beta}) = (k, k)$ on α-direction;
5: Apply Algorithm 2 with $(\underline{\alpha}, \underline{\beta}) = (k, k)$ on β-direction;
6: **return** κBCO-index;



(a) Peeling orders of Algorithm 1 (b) Peeling orders of Algorithm 5

Fig. 6. An example of different peeling orders in two index constructions for BCO-index and κBCO-index.

calculated by the $(\alpha - 1, \beta)$-core from the α-direction; or it can be calculated by the $(\alpha, \beta - 1)$-core from the β-direction. As a result, we can just keep one direction for a node to save lots of time and space as shown in the following lemma.

Lemma 2 (Tight Bound [24]). *For a node p, $\forall(\alpha, \beta) \in \Phi(p)$, the minimum value of (α, β) cannot be greater than the core number $\theta(p)$, i.e., $\theta(p) \geq \min(\alpha, \beta)$.*

Lemma 2 shows that given the core number $\theta(u)$ of node u, u cannot be in a (α, β)-core with $\alpha > \theta(u)$ and $\beta > \theta(u)$. Therefore, if we use the smaller value of (α, β) as array indices to store all pairs of $\Phi(u)$, the space cost is in $O(\theta(u))$.

The structure of κBCO-index. Based on this observation, we propose our improved index with a new search strategy of *κ-direction*, which treats the bipartite graph as a general graph and applies core decomposition, i.e., the case of $\alpha = \beta = k$.

Thus, we have $L^u_\kappa = L_{(\theta(u),\theta(u))\text{-core}\rightarrow(\theta(u)+1,\theta(u)+1)\text{-core}}(u)$.

Definition 10 (κBCO-index). *Given a node $u \in G$, κBCO-index(u) consists of bi-core numbers and onion layers on three directions: κ-direction with $\alpha = \beta$; α-direction with $\alpha > \beta$; and β-direction with $\alpha < \beta$, i.e.,* κBCO-index$(u) = \{ (\theta(u), L^u_\kappa), [(\alpha_u[i], L^u_\alpha((\alpha_u[i], i)))], [(\beta_u[j], L^u_\beta((j, \beta_u[j])))] \}, \forall 2 \leq i, j \leq \theta(u)$.

Table 1 shows an example of κBCO-index of Fig. 2. The first line is κBCO-index of node u_2, only considering all solid lines. The second line is the new κBCO-index of node u_2 after inserting the edge (u_2, v_1). The κBCO-index also contains the information of neighbors. The neighbors show the degree of the node, which can answer queries of $\alpha = 1$ or $\beta = 1$. Every node has a pair of integers on κ-direction, where the first one is the core number of the node and another is the onion layer. For example, $(1, 4)$ indicates $\theta(u_2) = 1$ and when $k = 1$, the onion layer of u_2 is 4. Every node p on α-direction or β-direction has $\theta(p) - 1$ ordered pairs. For example, the first pair of node u'_2 on β-direction is $(2, 1)$, indicates u'_2 is contained in the $(2, 1)$-core, onion layer is 1 and is not contained in the $(3, 1)$-core.

Algorithm 6 κBCO-index Maintenance for Batch Insertions

Require: Graph G, κBCO-index, an edge set ΔE to be inserted;
Ensure: κBCO-index;
1: $Candidate, S \leftarrow \emptyset$; Add ΔE to G;
2: **for all** $(u, v) \in \Delta E$ **do**
3: **for all** $(\alpha, \beta, L^u, direction) \in \kappa$BCO-index$(u)$ **do**
4: **if** $\epsilon(u) = false$ **then**
5: Push (α, β, L^u, u) into $Candidate_{direction}$;
6: Similar operations for node v, like lines 3-5;
7: **while** $Candidate_\kappa \neq \emptyset$ **do**
8: Find the smallest k in $Candidate_\kappa$;
9: Move nodes with the same k in $Candidate_\kappa$ to S;
10: Conduct Algorithm 3 on κ-direction to get κBCO-index and S^+;
11: **for all** $direction \in \{$"α", "β", "κ"$\}$ **do**
12: Add $(k+1, k+1, 1)$ to κBCO-index$_{direction}(p)$, $\forall p \in S^+$;
13: **for all** $p \in S^+ \cup N(S^+)$ **do**
14: **if** $\epsilon(p) = false$ **then**
15: Push $(k+1, k+1, 1, p)$ into $Candidate_{direction}$;
16: **while** $Candidate_\alpha \neq \emptyset$ **do**
17: **for all** β and α in $Candidate_\alpha$ in increasing order **do**
18: Move nodes with the same α and β in $Candidate_\alpha$ to S;
19: Conduct Algorithm 3 on α-direction to get κBCO-index and S^+;
20: Add $(\alpha+1, \beta, 1)$ to κBCO-index$_\alpha(p)$, $\forall p \in S^+$;
21: **for all** $p \in S^+ \cup N(S^+)$ **do**
22: **if** $\epsilon(p) = false$ **then**
23: Push $(\alpha+1, \beta, 1, p)$ into $Candidate_\alpha$;
24: Similar operations for $Candidate_\beta$, like lines 16-23;
25: **return** κBCO-index;

As shown in Fig. 5(a), we now conduct two Phase-I&II of κBCO-index indexing and maintenance. We construct κBCO-index by Algorithm 5, which invokes Algorithm 2 to peel G in three directions of α, β, and κ. Algorithm 5 shows the process of constructing κBCO-index. First of all, we conduct core decomposition on the graph (line 2) and get $k_{\max}$. Then, for each value of k (line 3), we conduct core decomposition on α-direction and β-direction. Finally, the result κBCO-index has been obtained. Fig. 6 shows the difference between two index construction algorithms. Algorithm 1 peels G from 1 to $\alpha_{\max}$ on β-direction and from 1 to $\beta_{\max}$ on α-direction. The total number of peeling process is $\alpha_{\max} + \beta_{\max}$. Algorithm 5 first peels G to compute all k-cores, also called peeling on κ-direction. After peeling all nodes in G, $k_{\max}$ is obtained. Then, it peels G from 2 to $k_{\max}$ on α-direction and from 2 to $k_{\max}$ on β-direction. The total number of peeling process is $2k_{\max} - 1$. We analyze the complexity of κBCO-index as follows.

Theorem 3. *The* κBCO-index *construction in Algorithm 5 takes* $O(mk_{\max})$ *time and* $O(K)$ *space, where* $K = \sum_{k=1}^{k_{\max}}(|U_k| + |V_k|)$ *and* $U_k \cup V_k$ *is the* k*-core of* G*. The* κBCO-index *size is* $O(K)$.

Proof. A peeling process takes $O(m)$ time. The total number of peeling by Algorithm 5 is $2k_{\max} - 1$. For any node p in G, the number of κBCO-index stored is $2\theta(p) - 1$. Thus, the space of κBCO-index is $O(\sum_{p \in U \cup V}(2\theta(p) - 1)) = O(\sum_{k=1}^{k_{max}}(|U_k| + |V_k|)) = O(K)$. Thus, Algorithm 5 takes $O(mk_{\max})$ time and $O(K)$ space. □

5.2 κBCO-Index Maintenance

Next, we introduce batch insertion/deletion to update κBCO-index in Algorithms 6 and 7, as shown in our framework of Fig. 5(a).

Maintain κBCO-index for batch edge insertion. Algorithm 6 shows how to maintain κBCO-index when inserting multiple edges into G in one batch. First, we insert all edges into G together and push all nodes and (α, β) that do not meet Property 2 into *Candidate* (lines 1-6). Since *Candidate* contains all nodes not meeting Property 2, the following process can maintain κBCO-index correctly. Then, we first maintain Φ on κ-direction (lines 7-15). We handle nodes in the increasing order of k, and every node increasing its (α, β) will be added new indexes on three directions (line 20). Since indexes changed, we put other affected nodes into *Candidate* (lines 13-15). Next, we maintain Φ on α-direction and β-direction (lines 16-24). These processes are similar to that of Algorithm 4.

Theorem 4. *Algorithm 6 takes $O(||\ \mathsf{L\_CHG}\ ||_1)$ time and uses $O(nk_{\max})$ space.*

Proof. As we prove in Theorem 2, Algorithm 3 only visits nodes in $||\mathsf{L\_CHG}||_1$. Algorithm 6 visits nodes of 1-hop neighborhood of inserted edges and the results of Algorithm 3, thus Algorithm 6 also takes $O(||\mathsf{L\_CHG}||_1)$ time. Then, Algorithm 6 keeps κBCO-index for all nodes in the graph, which takes $O(nk_{\max})$ space. □

Maintain κBCO-index for batch edge deletion. As mentioned before, the bi-core maintenance is bounded in case of deletion. Thus, we can simply search all nodes that changed their bi-core numbers Φ in BFS manner. In order to unify the index for insertion and deletion, after maintaining Φ, we also need to spend extra time to maintain onion layers. Therefore, Algorithm 7 for deletion maintenance will first maintain Φ (Algorithm 8) and then maintain onion layers (Algorithm 9). Both Algorithm 8 and Algorithm 9 search nodes in BFS manner. Algorithm 8 finds nodes that do not meet Definition 2, while Algorithm 9 finds nodes that do not meet Property 2. Algorithm 8 reduces α, β or θ of found nodes by one (lines 4-6), while Algorithm 9 estimates a smaller onion layer for the found nodes (line 3). In Algorithm 7, it first removes nodes and finds all candidates in one batch (lines 1-6). The next parts are similar to the Algorithm 6 for insertion maintenance. The differences are that Algorithm 7 processes nodes in the decreasing order of α, β or θ (line 8, 24). When a node decreases its θ in κ-direction, α and β in other two directions also needed to be removed (lines 14-22).

For example, consider the whole $(2,2)$-core in Fig. 2, we use Algorithm 7 to maintain κBCO-index when (u_5, v_4) is removed. First, we find $Candidate_\kappa = \{(2,2,1,u_5), (2,2,2,v_4)\}$, $Candidate_\alpha = \{(2,2,1,u_5)\}$ and $Candidate_\beta = \{(2,2,2,u_5), (2,2,3,v_4)\}$ (lines 1-6). Then, we prioritize processing $Candidate_\kappa$ (lines 7-22) and get $k = 2$ (line 8), $S_k = \{u_5\}$ (line 9) and $S = \{v_4\}$ (line 10). After applying Algorithm 8 on S_k, we get $S^- = \{u_3, u_4, u_5, v_3, v_4, v_5\}$ (line 11). In line 11, we push u_2 into S and $S = \{u_2, v_4\}$. Since v_4 is not in $(2,2)$-core, we only update $L_\kappa^{u_2}$ from 2 to 1 in line 13. Next, we find new candidates affected by the decrease of bi-core numbers of nodes in S^-.When all nodes in $Candidate_\kappa$ are handled, we continue to handle other two candidate sets.

Theorem 5. *Algorithm 7 is bounded by* L_CHG, *which takes $O(||\mathsf{L\_CHG}||_1)$ time and uses $O(K)$ space.*

Proof. In Algorithm 8, the nodes that changed bi-core numbers Φ are in L_CHG. Thus, S is bounded by L_CHG, and the visited nodes is bounded by the 1-hop neighborhood of L_CHG. Similarly, in Algorithm 9, nodes in S that change their onion layers, are in L_CHG. The visited nodes are bound by $O(||\mathsf{L\_CHG}||_1)$. As a result, Algorithm 7 invoking Algorithms 8 and 9 is bounded by L_CHG, which takes $O(||\mathsf{L\_CHG}||_1)$ time and $O(K)$ space. □

Algorithm 7 κBCO-index Maintenance for Batch Deletions

Require: Graph G, κBCO-index, an edge set ΔE to be removed;
Ensure: κBCO-index;
1: $Candidate, S \leftarrow \emptyset$; Remove ΔE from G;
2: **for all** $(u, v) \in \Delta E$ **do**
3: **for all** $(\alpha, \beta, L^u, direction) \in \kappa$BCO-index$(u)$ **do**
4: **if** $\epsilon(u) = false$ **then**
5: Push (α, β, L^u, u) into $Candidate_{direction}$;
6: Similar operations for node v, like lines 3-5;
7: **while** $Candidate_\kappa \neq \emptyset$ **do**
8: Move nodes with the largest k in $Candidate_\kappa$ to S;
9: Find set S_k in S that nodes in S_k violate Definition 2;
10: Set $S \leftarrow S - S_k$;
11: Call Algorithm 8 with input (κBCO-index, S_k, k, k, "κ") to get S^-;
12: Add p to S, where $p \in N(S^-)$ and $\epsilon(p) = false$;
13: Call Algorithm 9 with (κBCO-index, S, k, k, "κ") to update index;
14: **for all** $p \in S^-$ **do**
15: Push $(k-1, k-1, \infty, p)$ into $Candidate_\kappa$;
16: Set $\beta \leftarrow k$, $(\alpha, l) \leftarrow \kappa$BCO-index$_\alpha(p)[k]$;
17: Push (α, β, l, p) to $Candidate_\alpha$; Remove κBCO-index$_\alpha(p)[k]$;
18: **for all** $q \in N(p)$ **do**
19: **if** $\epsilon(q) = false$ **then**
20: Push (α, β, L^q, q) into $Candidate_\alpha$;
21: Set $\alpha \leftarrow k$, $(\beta, l) \leftarrow \kappa$BCO-index$_\beta(p)[k]$;
22: Similar operations for β-direction, like lines 17-20;
23: **while** $Candidate_\alpha \neq \emptyset$ **do**
24: **for all** β and α in $Candidate_\alpha$ in decreasing order **do**
25: Move nodes with the same α and β in $Candidate_\alpha$ to S;
26: Similar operations for α-direction, like lines 9-13;
27: Push $(\alpha - 1, \beta, \infty, p)$ into $Candidate_\alpha$, $\forall p \in S^-$, if $\alpha > \beta$;
28: Similar operations for $Candidate_\beta$, like lines 23-27;
29: **return** κBCO-index;

Algorithm 8 Core Number Maintenance for Edge Deletion

Require: κBCO-index, S, α, β, $direction$;
Ensure: κBCO-index, S^-;
1: $S^- \leftarrow \emptyset$;
2: **while** $S \neq \emptyset$ **do**
3: Move a node p from S to S^-, remove it from κBCO-index;
4: **if** $direction$ = "α" **then** Set κBCO-index$_\alpha(p)[\beta] \leftarrow (\alpha - 1, \infty)$;
5: **if** $direction$ = "β" **then** Set κBCO-index$_\beta(p)[\alpha] \leftarrow (\beta - 1, \infty)$;
6: **if** $direction$ = "κ" **then** Set $\theta(p) \leftarrow \alpha - 1$ and $L^p \leftarrow \infty$;
7: **for all** $q \in N(p)$ and $q \notin S$ **do**
8: Push q into S, if q violates Definition 2;
9: **return** κBCO-index, S^-;

A comparison of technical details and algorithm complexity between ours and [24].

- (1) Why the techniques of [24] cannot work on κBCO-index? Note that maintaining bi-core numbers and onion layers is different. The decrease of the bi-core number (α, β) of a node can affect its neighbors with the same bi-core number but cannot affect neighbors with bi-core

Algorithm 9 Onion Layer Maintenance for Edge Deletion

Require: Graph $G = (U, V, E)$, κBCO-index, α, β, $direction$, S;
Ensure: κBCO-index;
1: **while** $S \neq \emptyset$ **do**
2: Pop a node p in S with the smallest L^p;
3: Set $L^p \leftarrow l$, which is the largest value meeting Property 2;
4: **for all** $q \in N(p)$ and $q \notin S$ **do**
5: Add q to S, if $\epsilon(q) = false$;
6: **return** κBCO-index;

Algorithm 10 Constructing Bipartite Graph for k-(r, s)-nucleus

Require: General graph $\overline{G}$, two integers r, s with $r \leq s$;
Ensure: Bipartite graph $G = (U, V, E)$;
1: $R \leftarrow$ find all r-cliques in $\overline{G}$; $S \leftarrow$ find all s-cliques in $\overline{G}$;
2: $G \leftarrow (R, S, \emptyset)$;
3: **for** each s-clique $H_s \in S$ **do**
4: **for** each r-clique $H_r \subseteq H_s$ **do**
5: $id_r \leftarrow$ id of H_r in R, $id_s \leftarrow$ id of H_s in S;
6: Add edge (id_r, id_s) to G;
7: **return** G;

numbers $(\alpha + 1, \beta)$ or $(\alpha, \beta + 1)$. However, the decrease of the onion layer l of a node can affect its neighbors at the same onion layer l and a higher onion layer $l + 1$, which leads to the different techniques for maintaining bi-core numbers and onion layers. Therefore, the techniques of maintaining bi-core numbers in [24] cannot be applied on maintaining onion layers of our κBCO-index.

- (2) We analyze the updating complexity of edge insertions. For maintenance of edge insertion, our Algorithm 6 is bounded by taking $O(||\mathsf{L\_CHG}||_1)$ time, while as mentioned in [24], the BLI method [24] is unbounded by taking $O(md_{\max})$ time. In the worst case, BLI needs to visit the whole graph for each changed bi-core number, while Algorithm 6 only visits nodes that need to update onion layers.
- (3) We analyze the updating complexity of edge deletions. For maintenance of edge deletion, our Algorithm 7 is bounded by taking $O(||\mathsf{L\_CHG}||_1)$ time, and the BLD method [24] is unbounded by taking $O(||\mathsf{C\_CHG}||_1)$ time, where C_CHG is the changed part of bi-core numbers. If a node changes its bi-core numbers, its onion layers also change, thus $|\mathsf{C\_CHG}| \leq |\mathsf{L\_CHG}|$, indicating that Algorithm 7 may take more time than BLD. However, as our maintenance of onion layers is bounded for edge insertion, Algorithm 6 performs much better than BLI in experiments. Thanks to a huge efficiency improvement of updating for edge insertions, the extra cost for maintaining the structure of onion layers in case of edge deletion is worth.

5.3 Handle General Maintenance for k-Nucleus

We apply our algorithms to handle general maintenance for k-nucleus as shown in Fig. 5(a). Given a general graph $\overline{G}$, we develop Algorithm 10 to construct a bipartite graph G. When a general update $\Delta\overline{G}$ comes, we construct the corresponding update ΔG by Algorithm 11. Then, the maintenance can be done by the same bi-core techniques in previous sections. Other dense subgraph maintenance can be done similarly.

Algorithm 11 Update for Edge Insertions (Deletions)

Require: Original graph $\overline{G}$, an edge set $\Delta\overline{E}$ to be inserted (deleted), original bipartite graph $G = (U, V, E)$;
Ensure: Updated bipartite graph G;
1: $\overline{G}' \leftarrow \overline{G} \pm \Delta\overline{E}$;
2: $\Delta S \leftarrow$ all **new** (deleted) s-cliques in $\overline{G}'$ ($\overline{G}$) but not in $\overline{G}$ ($\overline{G}'$);
3: $\Delta R \leftarrow$ all **new** (deleted) r-cliques in $\overline{G}'$ ($\overline{G}$) but not in $\overline{G}$ ($\overline{G}'$);
4: Add vertices to G: $U \leftarrow U \cup \Delta S$, $V \leftarrow V \cup \Delta R$;
5: **for** each s-clique $H_s \in \Delta S$ **do**
6: **for** each r-clique $H_r \subseteq H_s$ **do**
7: $id_r \leftarrow$ id of H_r in U, $id_s \leftarrow$ id of H_s in V;
8: Add (Remove) edge (id_r, id_s) to (from) G;
9: (Remove ΔS and ΔR from G);
10: **return** G;

Algorithm 10 shows how to construct the bipartite graph G from the general graph $\overline{G}$ for maintaining the (r, s)-nucleus on $\overline{G}$ by following Section 3.2. First, we list all r-cliques and s-cliques in $\overline{G}$ (line 1). Then, for each pair containing relationships that a r-clique is contained in a s-clique, we push their ids, id_r and id_s, into G (lines 3-6). Finally, G is the constructed bipartite graph.

Lemma 3. *The $(k, \binom{s}{r})$-core of G is the k-(r, s)-nucleus of $\overline{G}$.*

Proof. Every node in $U(G)$ is a r-clique in $\overline{G}$. There are exact $\binom{s}{r}$ r-cliques in a s-clique. So if a node in $V(G)$ has degree less than $\binom{s}{r}$, the corresponding s-clique in $\overline{G}$ also disappears. Therefore, we can ensure that every node $v \in V(G)$ in the $(k, \binom{s}{r})$-core is a valid s-clique in $\overline{G}$, and every node $u \in U(G)$ in the $(k, \binom{s}{r})$-core is contained in at least k s-cliques. In conclusion, the $(k, \binom{s}{r})$-core of G is the k-(r, s)-nucleus of $\overline{G}$. □

The k-core and k-truss are two special cases of k-(r, s)-nucleus for different parameters r and s. Thus, we prove the equivariance of k-core and k-truss by Lemma 3 in the following corollary.

Corollary 1. *Given a general graph $\overline{G} = (\overline{V}, \overline{E})$, we construct a bipartite graph $G_1 = (V_1, U_1, E_1)$, where $V_1 = \overline{V}$, $U_1 = \{e = (u, v) \in \overline{E}\}$ and $E_1 = \{(p, e) : p \in \overline{V}, e \in \overline{E}, p \in e\}$. The $(k, 2)$-core of G_1 is the k-core of $\overline{G}$.*

Corollary 2. *Given a general graph $\overline{G} = (\overline{V}, \overline{E})$, we construct a bipartite graph $G_2 = (V_2, U_2, E_2)$, where $V_2 = \overline{E}$, $U_2 = \{\triangle_{vuw} : (u, v) \in \overline{E}, w \in N(u) \cap N(v)\}$ and $E_2 = \{(e, \triangle_{vuw}) : e \in \overline{E}, e \in \triangle_{vuw}\}$. The $(k-2, 3)$-core of G_2 is the k-truss of $\overline{G}$.*

The most time-consuming part is enumerating all s-cliques, so the time complexity is $O(nd_{\max}^{s-1})$. We need to store all r-cliques, s-cliques and their containing relationships, so the space complexity is $O(r|R| + s|S| + \binom{s}{r}|S|) = O(r|R| + \binom{s}{r}|S|)$, where $|R|$ and $|S|$ are the number of r-cliques and s-cliques, respectively. Once we obtain the bipartite graph G, we can construct κBCO-index accordingly.

When the general graph $\overline{G}$ changes, we can maintain the constructed bipartite graph G and κBCO-index, to replace directly maintaining the k-(r, s)-nucleus. Algorithm 11 shows the process of constructing the updated bipartite graph G. The green and underlined part is for edge insertions and the red part enclosed by parentheses is for edge deletions. First of all, we obtain the new graph $\overline{G}'$ (line 1). The edge insertions create new r-cliques and s-cliques. Each s-clique contains exactly $\binom{s}{r}$ r-cliques, so it cannot happen that a new r-clique connects to an old s-clique. Therefore, in line 5, we only need to search all new s-cliques. The deletion case is similar. Once a bipartite graph

Table 2. Network statistics

Network	|U|	|V|	|E|	d_{max}	k_{max}	K
IMDB (IMDB)	303,617	896,302	3,782,463	1,590	23	3,921,066
Wikipedia-en (WC)	1,853,493	182,947	3,795,796	11,593	18	4,001,271
amazon-ratings (AR)	2,146,057	1,230,915	5,743,258	12,180	26	6,115,584
dblp (DBLP)	1,953,085	5,624,219	12,282,059	1,386	14	13,597,946
Wikipedia-edits-de (DE)	1,025,084	5,910,432	55,231,903	818,183	212	55,485,022
Delicious tag–item (DTI)	4,511,972	33,777,768	137,240,382	1,057,753	180	138,505,211
Web trackers (WT)	27,665,730	12,756,244	140,613,762	11,571,952	437	142,828,289
bag-pubmed (BP)	8,200,000	141,043	483,450,157	2,323,263	108	483,958,627

G finishes the update, we can maintain κBCO-index accordingly by following Section 5. As a result, we can maintain the k-(r, s)-nucleus.

Discussion of parallelism updating. We explore the opportunity of parallelism updating by identifying independent operations. The maintenance on α-direction and on β-direction do not affect each other. Meanwhile, for different bi-core numbers, the maintenance can be processed efficiently in parallel. In addition, in our general maintenance framework, we can enumerate different sizes of cliques using parallel algorithms. All of these potential parallelism techniques provide further efficiency improvement of our solutions.

6 Experiments

In this section, we conduct experiments to evaluate the performance of our solutions. The tests are conducted on a Linux Server with Xeon Gold 6230R (2.1 GHz) and 768GB main memory. All algorithms are implemented in C++. The source code is publicly available. [1]

Datasets. We use 8 real-world bipartite graphs from KONECT [2] in the experiments. Table 2 reports the network statistics. We also report the maximum k values of the k-core as $k_{\max}$ and the total number of nodes of all k-cores as $K = \sum_{i=1}^{k_{\max}}(|U_k| + |V_k|)$, which is relevant to the size of our index κBCO-index.

Comparison methods. Different methods are compared below.

- BLI: The state-of-the-art bi-core maintenance algorithm for edge insertion in [24], which is unbounded.
- BLD: The state-of-the-art bi-core maintenance algorithm for edge deletion in [24], which is bounded.
- OLCI: Our bi-core maintenance method using Algorithm 6 for batch edge insertions, which is bounded by the 1-hop neighborhood of nodes that changed onion layers.
- OLCD: Our bi-core maintenance method using Algorithm 7 for batch edge deletion, which is bounded.

In addition, we evaluate the construction methods of BCO-index by Algorithm 1 and κBCO-index by Algorithm 5.

Evaluation metrics and parameters. The bi-core maintenance includes two parts: insertions and deletions. Following [36] to dismiss the bias of updating performance, we randomly insert/delete 100 edges into/from each dataset by default. In Exp-IV, we also test up to 10,000 updated edges. We first remove these edges from the dataset to evaluate the efficiency of deletion, and then insert them back to evaluate the efficiency of insertion.

[1] https://github.com/jinrdfh/bi-core
[2] http://konect.cc/networks/

Table 3. Efficiency evaluation of all maintenance algorithms. The running time is the total cost of 100 edge insertions or deletions in seconds. The best results are in bold.

	Index Time		Insertion		Deletion	
Graph	Index [24]	Ours	BLI [24]	OLCI	BLD [24]	OLCD
IMDB	12.75	4.24	27.65	**0.15**	**0.16**	0.24
WC	12.15	2.99	0.32	**0.077**	0.22	**0.07**
AR	67.36	5.59	12.85	**0.14**	0.4	**0.18**
DBLP	38.5	5.34	264.45	**0.031**	0.43	**0.01**
DE	726.27	246.63	3,957	**230.52**	**151.87**	221.17
DTI	1,895	795.91	934.22	**31.98**	140.43	**33.2**
WT	6,165	837	1,293	**260.35**	310.62	**168.46**
BP	1,398	3,104	81,628	**1,059**	**209.75**	1,163

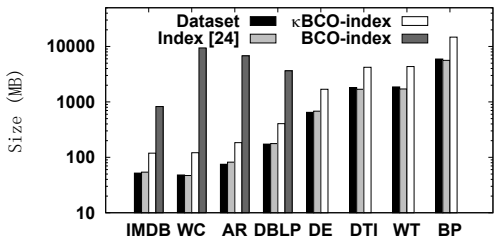


Fig. 7. Evaluation of the size of our index, BCO-index and κBCO-index, compared with the dataset size and the index size of [24]. Blank values mean the space exceeds 100 GB.

Exp-I: Efficiency evaluation. We first evaluate the efficiency of our proposed algorithm and competitors through inserting/deleting 100 edges, as shown in Table 3, together with the indexing time. We have the following observations. First of all, our insertion algorithm OLCI is much faster than baseline insertion algorithm BLI on all datasets, especially on the dataset DBLP, which achieves 8530**x** speedup. Secondly, our deletion algorithm OLCD runs faster than the baseline deletion algorithm BLD on most datasets, and their running time is of the same order of magnitude. As we mentioned earlier, OLCD is bounded by the nodes that changed onion layers, and BLD is bounded by the nodes that changed Φ. The result shows that the sizes of these two node sets are very close. On datasets IMDB, DE and BP, BLD is slightly faster than OLCD, but BLI is much slower than OLCI, showing that our extra cost of maintaining onion layers is worth. Third, the running times of OLCI and OLCD are also on the same order of magnitude, showing that both maintenance algorithms are bounded by nodes that changed onion layers. Finally, in terms of index time, our maintenance algorithms also have significant advantages on most datasets.

Exp-II: Index size evaluation. Fig. 7 reports the size of κBCO-index on all datasets, compared with the original dataset size. The data labeled by "κBCO-index" is the index size constructed by Algorithm 5 in Section 5, while the data labeled by "BCO-index" is the index size constructed by Algorithm 1 in Section 4. We also report the index size of [24], which is almost the same with the original dataset size. The program is forced to terminate if the space exceeds 100 GB. Algorithm 1 cannot finish the index construction within the given space on datasets DE, DTI, WT and BP. Algorithm 1 takes $O(nd_{\max})$ space, while Algorithm 5 takes $O(K)$ space, where $K = \sum_{k=1}^{k_{\max}} |U_k| + |V_k| \leq nk_{\max}$, $k_{\max} \ll d_{\max}$, as shown in Table 2. Thus, the index size of Algorithm 5 is much smaller than that of Algorithm 1, which shows a huge improvement by κBCO-index. In most cases, κBCO-index is only about twice as large as the original dataset, which is practically acceptable in real applications. The size of κBCO-index is also about twice as large as that of the

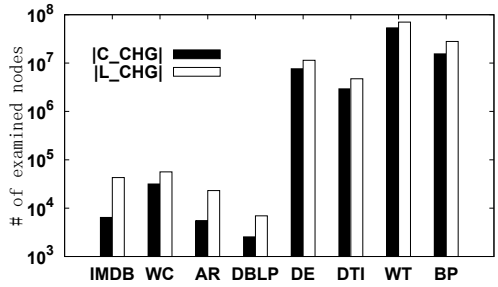
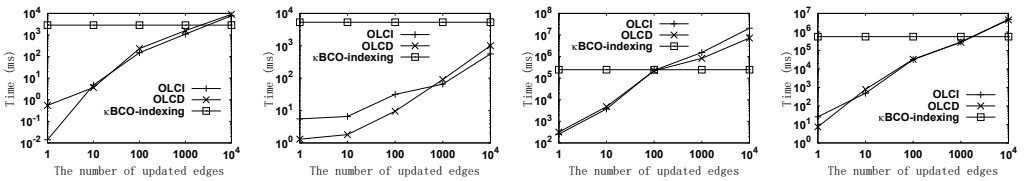


Fig. 8. The number of examined nodes. |C_CHG| is the number of changed Φ. |L_CHG| is the number of changed onion layers.

Table 4. Efficiency evaluation of general dense subgraph maintenance. All times are reported in milliseconds.

	General Graphs $\overline{G}$			Bipartite Graphs G	
Dense Subgraph Pattern	k-core	k-truss	k-$(3,4)$-nucleus	(α,β)-core	k-bitruss
U	node ○	edge ○—○	triangle △	node ○	edge ○—○
V	edge ○—○	triangle △	4-clique ⊠	node ○	butterfly ⋈
|U|	684, 912	2, 284, 991	4, 582, 169	1, 953, 085	12, 282, 059
|V|	2, 284, 991	4, 582, 169	26, 347, 208	5, 624, 219	31, 673, 959
|E|	4, 569, 982	13, 746, 507	105, 388, 832	12, 282, 059	126, 695, 836
ΔE	200	1, 968	35, 948	100	5, 728
Index Time	2, 550	5, 541	36, 217	5, 338	69, 994
OLCI	**20.21**	**268.03**	**15, 816**	**31.49**	**309.26**
OLCD	**20.20**	**414.71**	**20, 425**	**9.52**	**436.67**



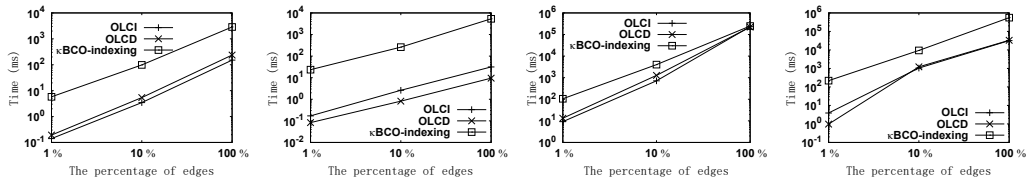
(a) IMDB (b) DBLP (c) DE (d) DTI

Fig. 9. The running time by varying the number of updated edges.

index of [24], because the index of [24] only keeps bi-core numbers of all nodes, while κBCO-index stores bi-core numbers, onion layers and neighbors of all nodes.

Exp-III: Evaluation of examined nodes. Fig. 8 reports the number of examined nodes during the process of OLCD. |C_CHG| is the number of changed Φ, which is a lower bound for any maintenance algorithm based on bi-core numbers. |L_CHG| is the number of changed onion layers, which is a lower bound for our maintenance algorithms. In most cases, |L_CHG| is about 3 times larger than |C_CHG|, showing that maintaining onion layers to replace maintaining Φ is practicable.

Exp-IV: Evaluation of the number of updated edges. We randomly select 1, 10, 100, 1000, 10^4 edges from datasets, IMDB, DBLP, DE and DTI, to test the efficiency of insertion and deletion. Fig. 9 reports the running time of our algorithms by varying the number of updated edges. When the update size is small, i.e., no more than 100, our algorithms can efficiently update onion layers in



(a) IMDB (b) DBLP (c) DE (d) DTI

Fig. 10. The running time by varying the number of edges in graph G.

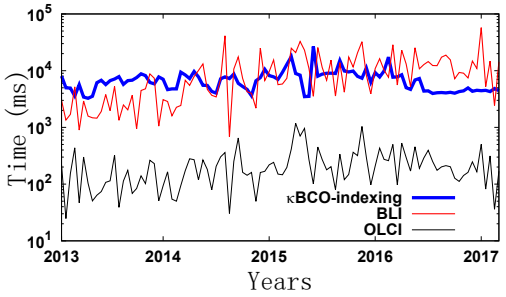


Fig. 11. Case study of bi-core maintenance on dynamic DBLP in 2013-2017, where the update takes every two weeks.

batches. The increase of running time is slow, indicating that updating in batches is more efficient than edge-by-edge maintenance. When the update size is large, i.e., 10^4, reconstruction of index is more efficient. This experiment shows that our maintenance algorithms are more efficient than rebuilding the index even when the number of updated edges reaches 10^3 (0.03%) on IMDB, 10^4 (0.8%) on DBLP, 100 (0.0002%) on DE and 10^3 (0.0007%) on DE. Our maintenance algorithms perform better on the dataset DBLP, because the $k_{\max}$ on DBLP is smaller, so fewer Φ need to be maintained.

Exp-V: Scalability test. We randomly select 1%, 10%, 100% edges from datasets, IMDB, DBLP, DE and DTI, to form new graphs. Then, we conduct experiments on new graphs with the same setting of Exp-I. Fig. 10 reports the running time of our algorithms and index construction. The results show that our maintenance algorithms have good scalability. The running time is almost linear with the scale of the graph.

Exp-VI: Bi-core maintenance in a practical application. We collect a collaboration network of DBLP containing the information of authors and publications in 2013-2017. We use the author and the corresponding publication to construct a bipartite graph. We use the data before 2013 to construct κBCO-index and use the later data to maintain κBCO-index every two weeks. The graph initially has $1,014,958$ edges with an average of $10,414$ edges inserted each time. Fig. 11 reports the running time of index construction, baseline maintenance algorithm BLI and our maintenance algorithm OLCI. BLI can initially be faster than rebuilding the index. However, with the graph size increasing, it gradually becomes less effective than rebuilding the index. Our algorithm is always about one order of magnitude faster than other algorithms, which indicates that our algorithm has significant advantages in real-world applications.

Exp-VII: General maintenance to handle k-core, k-truss, k-nucleus, and bi-truss. We use our bi-core maintenance algorithms to maintain k-core, k-truss, and k-$(3,4)$-nucleus in Table 4. The

Table 5. The running time of simultaneously maintaining k-core, k-truss, and (α, β)-core is reported in milliseconds.The competitors are state-of-the-art algorithms k-core [50], k-truss [36], and (α, β)-core [24].

	k-core	k-truss	(α, β)-core	3-in-1 maintenance task
SOTA	**15**	**50**	264, 880	264, 945
Ours	40	683	**41**	**764**

general graph $\overline{G}$ is a DBLP dataset with 684, 912 nodes and 2, 284, 991 edges, where the nodes are authors and the edge represents the co-authorship between two nodes. We insert/delete 100 edges. We construct bipartite graphs as mentioned in Section 3. Table 4 reports that our maintenance algorithms run faster than the index construction from scratch. And the speedup reaches 126**x** for k-core, 16**x** for k-truss, 2**x** for k-$(3, 4)$-nucleus, where the speedup is calculated by the index time over the average of insertion time and deletion time. Experimental results show that our algorithms can efficiently maintain k-nucleus. Note that with the increase of r and s, the bipartite graph size and ΔE also increase quickly. Therefore, the running time also increases quickly with the increase of r and s. In addition, we also test our maintenance algorithms to maintain k-bitruss. In this setting, the bipartite graph G of DBLP dataset keeps only the author-paper relationships. We construct a new bipartite graph following Section 3 and compute $(k, 4)$-cores for maintaining k-bitruss. Experimental results show that our algorithms can efficiently maintain k-bitruss and achieve 188**x** speedup than reconstructing the index.

In addition, we test a comprehensive and challenging 3-in-1 maintenance task by combining three tasks of maintaining k-core, k-truss and (α, β)-core into one, simultaneously. We report the total running time of inserting 100 edges into the graph and then removing these edges from the graph, which involves both edge insertions and edge deletions. In addition, we also report the individual running time of three state-of-the-art competitors k-core [50], k-truss [36], and (α, β)-core [24]. In terms of individual times, the SOTA competitors of k-core [50] and k-truss [36] run faster than ours, and the SOTA (α, β)-core competitor [24] runs much worse than ours. The reasons are two-fold. First, the maintenance algorithms [50] and [36] are designed specifically for one particular pattern, whereas ours is a unified maintenance framework for handling multiple general patterns. Second, our method needs to convert the original graph to a bipartite graph, which takes an extra cost of computations and space storage as shown in Table 4. Third, considering the comprehensive 3-in-1 task, our method wins an integrated method of these SOTA competitors in an easy implementation way, shown to be more useful in practice. In summary, our proposed unified framework can play an important role in one-off indexing scheme to simultaneously update multiple dense subgraph indexes over dynamic graphs, e.g., the N-in-1 maintenance task.

7 Conclusion

In this work, we creatively propose a unified framework to maintain various subgraphs (k-core, k-truss, k-nucleus, and k-bitruss) over dynamic graphs, which leverages our developed techniques of fast bi-core maintenance on bipartite graphs. We design a novel data structure of κBCO-index, which is compact to keep (α, β)-cores onion layers in three directions. We theoretically analyze the algorithm complexities and show the bounded advantage of our solutions. To our best knowledge, we are the first to successfully support the maintenance of k-nucleus and k-bitruss. Experimental results on show that our algorithms can be orders of magnitude faster than state-of-the-art competitors on batch updates.

Acknowledgments

This work is supported by grants from the Research Grants Council of the Hong Kong Special Administrative Region, China (Nos. HKBU 12201923, C2003-23Y, CUHK 14217622, and 14206625). Xin Huang is the corresponding author.

References

[1] Vladimir Batagelj and Matjaz Zaversnik. 2003. An O(m) Algorithm for Cores Decomposition of Networks. *CoRR* cs.DS/0310049 (2003).

[2] Monika Cerinsek and Vladimir Batagelj. 2015. Generalized two-mode cores. *Soc. Networks* 42 (2015), 80–87.

[3] Lijun Chang. 2023. Efficient Maximum k-Defective Clique Computation with Improved Time Complexity. *Proc. ACM Manag. Data* 1, 3 (2023), 209:1–209:26.

[4] Chen Chen, Mengqi Zhang, Renjie Sun, Xiaoyang Wang, Weijie Zhu, and Xun Wang. 2022. Locating pivotal connections: the K-Truss minimization and maximization problems. *WWW* (2022), 899–926.

[5] Jiujian Chen, Kai Wang, Ronghua Li, Hongchao Qin, Xuemin Lin, and Guoren Wang. 2024. Maximal Biclique Enumeration: A Prefix Tree Based Approach. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*. IEEE, 2544–2556.

[6] Kaiyu Chen, Dong Wen, Hanchen Wang, Zhengyi Yang, Wenjie Zhang, and Xuemin Lin. 2025. Covering K-Cliques in Billion-Scale Graphs. In *Proceedings of the ACM on Web Conference 2025, WWW 2025, Sydney, NSW, Australia, 28 April 2025- 2 May 2025*. 2299–2308.

[7] Qihao Cheng, Da Yan, Tianhao Wu, Lyuheng Yuan, Ji Cheng, Zhongyi Huang, and Yang Zhou. 2025. Efficient Enumeration of Large Maximal k-Plexes. In *Proceedings 28th International Conference on Extending Database Technology, EDBT 2025*. 53–65.

[8] J. Cohen. 2008. *Trusses: Cohesive Subgraphs for Social Network Analysis*. Technical Report. National Security Agency.

[9] Qiangqiang Dai, Ronghua Li, Donghang Cui, and Guoren Wang. 2024. Theoretically and Practically Efficient Maximum Defective Clique Search. *Proc. ACM Manag. Data* 2, 4 (2024), 206:1–206:27.

[10] Qingshuai Feng, You Peng, Wenjie Zhang, Xuemin Lin, and Ying Zhang. 2024. DSPC: Efficiently Answering Shortest Path Counting on Dynamic Graphs. In *Proceedings 27th International Conference on Extending Database Technology, EDBT 2024*. 116–128.

[11] Shuohao Gao, Kaiqiang Yu, Shengxin Liu, Cheng Long, and Zelong Qiu. 2024. On Searching Maximum Directed (k, ℓ)-Plex. In *40th IEEE International Conference on Data Engineering, ICDE 2024*. 2570–2583.

[12] Christos Giatsidis, Dimitrios M. Thilikos, and Michalis Vazirgiannis. 2011. D-cores: Measuring Collaboration of Directed Graphs Based on Degeneracy. In *11th IEEE International Conference on Data Mining, ICDM 2011*. IEEE Computer Society, 201–210.

[13] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying k-truss community in large and dynamic graphs. In *SIGMOD*. 1311–1322.

[14] Xin Huang, Laks V. S. Lakshmanan, Jeffrey Xu Yu, and Hong Cheng. 2015. Approximate Closest Community Search in Networks. *Proc. VLDB Endow.* 9, 4 (2015), 276–287.

[15] Yuli Jiang, Xin Huang, and Hong Cheng. 2021. I/O efficient k-truss community search in massive graphs. *VLDB J.* 30, 5 (2021), 713–738.

[16] Tuukka Korhonen, Wojciech Nadara, Michal Pilipczuk, and Marek Sokolowski. 2024. Fully dynamic approximation schemes on planar and apex-minor-free graphs. In *Proceedings of the 2024 ACM-SIAM Symposium on Discrete Algorithms, SODA 2024*. 296–313.

[17] Rong-Hua Li, Jeffrey Xu Yu, and Rui Mao. 2014. Efficient Core Maintenance in Large Dynamic Graphs. *IEEE Trans. Knowl. Data Eng.* 26, 10 (2014), 2453–2465.

[18] Xuankun Liao, Qing Liu, Xin Huang, and Jianliang Xu. 2024. Truss-based Community Search over Streaming Directed Graphs. *Proc. VLDB Endow.* 17, 8 (2024), 1816–1829.

[19] Xuankun Liao, Qing Liu, Jiaxin Jiang, Xin Huang, Jianliang Xu, and Byron Choi. 2022. Distributed D-core Decomposition over Large Directed Graphs. *Proc. VLDB Endow.* 15, 8 (2022), 1546–1558.

[20] Fengnian Lin, Boyu Ruan, Junhao Gan, and Lei Li. 2025. Efficient Bitruss Decomposition without Butterfly Enumeration. In *SIGKDD*.

[21] Boge Liu, Fan Zhang, Wenjie Zhang, Xuemin Lin, and Ying Zhang. 2021. Efficient Community Search with Size Constraint. In *ICDE*. 97–108.

[22] Qing Liu, Xuankun Liao, Xin Huang, Jianliang Xu, and Yunjun Gao. 2023. Distributed (α, β)-Core Decomposition over Bipartite Graphs. In *39th IEEE International Conference on Data Engineering, ICDE 2023*. 909–921.

[23] Qing Liu, Minjun Zhao, Xin Huang, Jianliang Xu, and Yunjun Gao. 2020. Truss-based Community Search over Large Directed Graphs. In *SIGMOD*. 2183–2197.

[24] Wensheng Luo, Qiaoyuan Yang, Yixiang Fang, and Xu Zhou. 2023. Efficient Core Maintenance in Large Bipartite Graphs. *Proceedings of the ACM on Management of Data* 1, 3 (2023), 1–26.

[25] Shohei Matsugu, Suomi Kobayashi, and Hiroaki Shiokawa. 2024. An Efficient Indexing Method for Dynamic Graph kNN. In *Database and Expert Systems Applications - 35th International Conference, DEXA 2024, Proceedings, Part I*, Vol. 14910. Springer, 81–89.

[26] Michael Mitzenmacher, Jakub Pachocki, Richard Peng, Charalampos E. Tsourakakis, and Shen Chen Xu. 2015. Scalable Large Near-Clique Detection in Large-Scale Networks via Sampling. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 815–824.

[27] Robert J Mokken. 1979. Cliques, clubs and clans. *Quality & Quantity* 13, 2 (1979), 161–173.

[28] Jiayang Pang, Chenhao Ma, and Yixiang Fang. 2024. A Similarity-based Approach for Efficient Large Quasi-clique Detection. In *Proceedings of the ACM on Web Conference 2024, WWW 2024*. 401–409.

[29] Serafeim Papadias, Zoi Kaoudi, Varun Pandey, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2024. Counting Butterflies in Fully Dynamic Bipartite Graph Streams. In *40th IEEE International Conference on Data Engineering, ICDE 2024*. 2917–2930.

[30] Jian Pei, Daxin Jiang, and Aidong Zhang. 2005. Mining Cross-Graph Quasi-Cliques in Gene Expression and Protein Interaction Data. In *ICDE*. 353–354.

[31] G. Ramalingam and Thomas W. Reps. 1996. On the Computational Complexity of Dynamic Graph Problems. *Theor. Comput. Sci.* (1996), 233–277.

[32] Seyed-Vahid Sanei-Mehri, Apurba Das, Hooman Hashemi, and Srikanta Tirthapura. 2021. Mining Largest Maximal Quasi-Cliques. *ACM Trans. Knowl. Discov. Data* 15, 5 (2021), 81:1–81:21.

[33] Ahmet Erdem Sariyüce, C. Seshadhri, Ali Pinar, and Ümit V. Çatalyürek. 2015. Finding the Hierarchy of Dense Subgraphs using Nucleus Decompositions. In *Proceedings of the 24th International Conference on World Wide Web, WWW 2015*. ACM, 927–937.

[34] Stephen B Seidman and Brian L Foster. 1978. A graph-theoretic generalization of the clique concept*. *Journal of Mathematical sociology* 6, 1 (1978), 139–154.

[35] Xin Sun, Xin Huang, Zitan Sun, and Di Jin. 2021. Budget-constrained Truss Maximization over Large Graphs: A Component-based Approach. In *CIKM '21: The 30th ACM International Conference on Information and Knowledge Management*. 1754–1763.

[36] Zitan Sun, Xin Huang, Qing Liu, and Jianliang Xu. 2023. Efficient Star-based Truss Maintenance on Dynamic Graphs. *Proc. ACM Manag. Data* (2023), 133:1–133:26.

[37] Zitan Sun, Xin Huang, Chengzhi Piao, Cheng Long, and Jianliang Xu. 2024. Adaptive Truss Maximization on Large Graphs: A Minimum Cut Approach. In *40th IEEE International Conference on Data Engineering, ICDE 2024*. 3270–3282.

[38] Zitan Sun, Xin Huang, Jianliang Xu, and Francesco Bonchi. 2021. Efficient Probabilistic Truss Indexing on Uncertain Graphs. In *WWW '21: The Web Conference 2021*. 354–366.

[39] Zitan Sun, Xin Huang, Jianliang Xu, Francesco Bonchi, and Lijun Chang. 2025. Probabilistic Truss Decomposition on Uncertain Graphs: Indexing and Dynamic Maintenance. *ACM Trans. Database Syst.* 50, 2, Article 6 (May 2025), 40 pages.

[40] Charalampos E. Tsourakakis. 2015. The K-clique Densest Subgraph Problem. In *Proceedings of the 24th International Conference on World Wide Web, WWW 2015, Florence, Italy, May 18-22, 2015*. 1122–1132.

[41] Jia Wang and James Cheng. 2012. Truss Decomposition in Massive Networks. *Proc. VLDB Endow.* 5, 9 (2012), 812–823.

[42] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. 2022. Towards efficient solutions of bitruss decomposition for large-scale bipartite graphs. *VLDB J.* 31, 2 (2022), 203–226.

[43] Kaixin Wang, Kaiqiang Yu, and Cheng Long. 2024. Efficient k-Clique Listing: An Edge-Oriented Branching Strategy. *Proc. ACM Manag. Data* 2, 1 (2024), 7:1–7:26.

[44] Bohua Yang, Dong Wen, Lu Qin, Ying Zhang, Lijun Chang, and Rong-Hua Li. 2019. Index-Based Optimal Algorithm for Computing K-Cores in Large Uncertain Graphs. In *ICDE*. 64–75.

[45] ZhiBang Yang, Xiaoxue Li, Xu Zhang, Wensheng Luo, and Kenli Li. 2022. K-truss community most favorites query based on top-t. *WWW* (2022), 949–969.

[46] Xiaowei Ye, Rong-Hua Li, Qiangqiang Dai, Hongchao Qin, and Guoren Wang. 2023. Efficient Biclique Counting in Large Bipartite Graphs. *Proc. ACM Manag. Data* 1, 1 (2023), 78:1–78:26.

[47] Haiyuan Yu, Alberto Paccanaro, Valery Trifonov, and Mark Gerstein. 2006. Predicting interactions in protein networks by completing defective cliques. *Bioinform.* 22, 7 (2006), 823–829.

[48] Ting Yu, Ting Jiang, Mohamed Jaward Bah, Chen Zhao, Hao Huang, Mengchi Liu, Shuigeng Zhou, Zhao Li, and Ji Zhang. 2024. Incremental Maximal Clique Enumeration for Hybrid Edge Changes in Large Dynamic Graphs. *IEEE Trans. Knowl. Data Eng.* 36, 4 (2024), 1650–1666.

[49] Yikai Zhang and Jeffrey Xu Yu. 2019. Unboundedness and Efficiency of Truss Maintenance in Evolving Graphs. In *SIGMOD*. 1024–1041.

[50] Yikai Zhang, Jeffrey Xu Yu, Ying Zhang, and Lu Qin. 2017. A Fast Order-Based Approach for Core Maintenance. In *ICDE 2017*. 337–348.

[51] Yikai Zhang, Jeffrey Xu Yu, Ying Zhang, and Lu Qin. 2023. Maintaining Top-t Cores in Dynamic Graphs. *IEEE Transactions on Knowledge and Data Engineering* 36, 9 (2023), 4766–4780.

[52] Feng Zhao and Anthony K. H. Tung. 2012. Large Scale Cohesive Subgraphs Discovery for Social Network Visual Analysis. *Proc. VLDB Endow.* 6, 2 (2012), 85–96.
[53] Weijie Zhu, Mengqi Zhang, Chen Chen, Xiaoyang Wang, Fan Zhang, and Xuemin Lin. 2019. Pivotal Relationship Identification: The K-Truss Minimization Problem. In *IJCAI*. 4874–4880.

Received July 2025; revised October 2025; accepted November 2025