Order-based Algorithms for Efficient Core Maintenance in Large Bipartite Graphs

QIAOYUAN YANG*, School of Computer Science, Peking University, China
WENSHENG LUO*†, College of Computer Science and Electronic Engineering, Hunan University, China
YIXIANG FANG†, School of Data Science, The Chinese University of Hong Kong, Shenzhen, China
YUANYUAN ZENG, School of Data Science, The Chinese University of Hong Kong, Shenzhen, China

The (α, β)-core, a.k.a. bi-core, is a fundamental model in bipartite graphs, extensively applied in various real-world applications such as product recommendation, fraud detection, and community detection. The dynamic nature of bipartite graphs, with frequent insertions and deletions of vertices and edges, makes maintaining bi-cores computationally expensive. Although recent work has addressed bi-core maintenance in dynamic bipartite graphs, existing approaches lack theoretical analysis regarding the changes in bi-cores corresponding to graph modifications, and also struggle with scalability and frequency of updates. To tackle these challenges, we systematically analyze and present bi-core maintenance algorithms with theoretical guarantees. Specifically, we conduct boundedness analysis, a key tool for analyzing incremental algorithms over dynamic graphs, on the bi-core maintenance problem. Our theoretical analysis shows that while the bi-core maintenance problem stays bounded under edge deletions, it becomes unbounded when handling edge insertions. To handle this unboundedness, we propose a novel structure called the BD-Order, which transforms the solution into a near bounded one. By leveraging the BD-Order, we introduce a novel order-based maintenance algorithm that effectively reduces the scope of affected vertices, thus enhancing efficiency. Additionally, the algorithm remains bounded for edge deletions through an auxiliary structure. The comprehensive experimental results on diverse real and synthetic datasets underscore the superior performance of our algorithms. Particularly, they achieve speed enhancements of up to two orders of magnitude over the state-of-the-art approaches.

CCS Concepts: • **Mathematics of computing** → **Graph algorithms**.

Additional Key Words and Phrases: Bipartite graphs, (α, β)-core, core maintenance, dynamic graphs

1 Introduction

The bipartite graph stands as a cornerstone in graph theory, comprising two distinct sets of vertices where edges exclusively link vertices from different sets. This concept serves as a powerful tool for modeling relationships across various real-world domains, including recommendation networks [10], collaboration networks [7], and gene co-expression networks [23]. For example,

---

*Both authors contributed equally to this research.
†Corresponding authors

Authors' Contact Information: Qiaoyuan Yang, 2401111962@stu.pku.edu.cn, School of Computer Science, Peking University, Beijing, China; Wensheng Luo, College of Computer Science and Electronic Engineering, Hunan University, Changsha, China, luowensheng@hnu.edu.cn; Yixiang Fang, School of Data Science, The Chinese University of Hong Kong, Shenzhen, Shenzhen, China, fangyixiang@cuhk.edu.cn; Yuanyuan Zeng, School of Data Science, The Chinese University of Hong Kong, Shenzhen, Shenzhen, China, zengyuanyuan@cuhk.edu.cn.
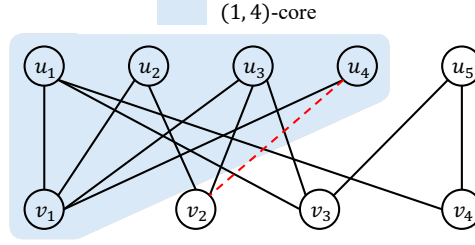
Fig. 1. A bipartite graph G, and (u_4, v_2) is an edge to be inserted (or deleted).

in recommendation networks, users and items typically constitute two distinct vertex types. The interactions between these vertices form a bipartite graph, with edges representing the historical purchase behaviors of users. In the realm of bipartite graph analysis, researchers have extensively investigated various cohesive subgraphs. Noteworthy examples include (α, β)-core [5, 26, 30], bitruss [42], and biclique [31]. Among these, the (α, β)-core, a.k.a. bi-core, has attracted considerable attention. In a bipartite graph $G = (U, V, E)$, where U and V are two disjoint sets of vertices, the (α, β)-core of G is the maximal subgraph where each vertex in U has at least α neighbors and each vertex in V has at least β neighbors. Figure 1 depicts a bipartite graph, where the edge (u_4, v_2) is temporarily excluded for the current illustration. The induced subgraph of vertices $\{u_1, u_2, u_3, u_4, v_1\}$ is the $(1, 4)$-core since the degrees of vertices in U are at least 1 and the degree of vertices in V are at least 4. Bi-core plays a crucial role in many applications, such as graph visualization [1], e-commerce recommendations [15], and fraud detection in financial systems [3, 7, 44].

Efficient computation of all bi-cores in a bipartite graph, known as bi-core decomposition, has been widely studied in recent years [9, 15, 22, 25, 41]. Although these methods achieve remarkable efficiency, they mainly target static graphs. In real-world scenarios, however, bipartite graphs evolve continuously as new entities and interactions appear or disappear. For example, Amazon Logistics processed 4.79 billion U.S. delivery orders in 2022, averaging over 13 million per day[1], forming a dynamic bipartite graph where edges represent user–producer interactions. Such large-scale dynamics cause frequent changes in bi-core structures, which can in turn affect downstream applications that rely on them. In collaborative platforms such as Wikipedia, editors and articles form a bipartite graph where edges denote edit operations, and maintaining bi-cores helps identify dense editor–article communities useful for recommendation and anomaly detection. Therefore, it is practically important to develop algorithms that can efficiently maintain bi-cores under graph updates, a task known as *bi-core maintenance*. This work focuses on the single-edge update scenario, which serves as the foundation for more complex cases such as batch updates or restricted maintenance within specific (α, β) ranges. Although real systems may not update bi-cores after every single change, efficiently handling this fundamental case is essential for preserving bi-core structures in dynamic bipartite graphs.

Bi-core maintenance finds various applications across different domains: *(1) E-commerce Recommendations:* Platforms like Amazon rely on user-product interactions, forming a dynamic bipartite graph [15, 19, 33]. Timely updates to bi-cores are crucial for delivering relevant product suggestions and ensuring a satisfactory shopping experience. *(2) Fraud Detection in Financial Systems:* Financial platforms such as PayPal model fraudster networks and their controlled accounts as bi-core structures within bipartite graphs [3, 7, 44]. Swift updates to these bi-cores are essential for promptly identifying and mitigating fraudulent activities, thereby preventing financial losses. *(3) Dynamic Graph Analysis:* Bi-core maintenance serves as a fundamental component for various computational

---

[1]https://capitaloneshopping.com/research/amazon-logistics-statistics/

problems in dynamic bipartite graphs. This includes identifying bicliques [28, 32, 48], computing the maximum k-biplex [29], and analyzing the hierarchical structure of bipartite graphs [43]. Efficient bi-core maintenance enhances the effectiveness of analysis and problem-solving in these bipartite networks.

A naive algorithm of bi-core maintenance recomputes all bi-cores from scratch whenever an edge is inserted or deleted. This, however, is costly, requiring $O(m\sqrt{m})$ time [25], where m is the number of edges in the graph. In real-world scenarios, the number of changed vertices and edges is typically much smaller than the size of the entire graph. For instance, English Wikipedia experienced only a 2.6% increase in articles in 2023[2]. Given such limited changes, the corresponding updates to bi-cores are also expected to be minor. Leveraging this insight, Liu et al. [26] proposed an algorithm that focuses on local subgraphs near the inserted or deleted edges, achieving significant improvements over previous methods. Although this approach reduces redundant computation, it still involves exhaustive bi-core checks. To further optimize efficiency, Luo et al. [30] introduced the notion of bi-core numbers by exploiting the partially nested structure of bi-cores. The algorithm refines the update scope by analyzing the effect of edge insertions and deletions on vertex bi-core numbers, thus mitigating unnecessary computation and enhancing performance. However, several limitations persist. These algorithms still rely on breadth-first search (BFS) to traverse potentially affected vertices, leading to redundant exploration. Due to widespread update propagation among high-degree vertices, they also suffer from inefficiencies when processing edge insertions in dense subgraphs. Moreover, the absence of theoretical bounds on the number of visited vertices during updates leaves the maintenance cost analytically uncertain. Therefore, there remains substantial room for improving both the theoretical guarantees and algorithmic efficiency of bi-core maintenance.

Our key contributions. In this paper, we aim to address the aforementioned issues by proposing efficient bi-core maintenance algorithms that offer a theoretical guarantee regarding the scope of visited vertices.

(1) Boundedness analysis. As a crucial tool for analyzing incremental computations on dynamic graphs, the boundedness analysis has been widely used for analyzing maintenance algorithms [16]. *A maintenance algorithm is considered bounded if its cost is polynomial to graph updates and result changes; otherwise, it is deemed unbounded.* We conduct a detailed theoretical analysis of the boundedness of bi-core maintenance algorithms, and the analysis reveals that while maintaining all bi-cores in a bipartite graph with edge insertions leads to unboundedness, maintaining them with edge deletions results in boundedness. While bi-core maintenance with edge insertions is an unbounded problem, it is still possible to assess the required cost of the maintenance algorithm. Fan et al. [16, 17] propose a new metric termed *relative boundedness*, which quantifies the necessary cost for incrementalizing the recomputation algorithm in terms of changes in input data, output data, and auxiliary structures. Zhang et al. [49] propose a similar metric *near boundedness*, which is better suited for asymmetric problems (i.e., problems that are unbounded for either edge insertions or deletions and are bounded for the other). An unbounded incremental problem is still considered efficiently solvable if a relatively bounded or near bounded incremental algorithm exists [16, 17, 49]. Therefore, to efficiently maintain bi-cores, our objective is to develop a near bounded algorithm for the case of edge insertions (due to the asymmetry of bi-core maintenance algorithm), and a bounded algorithm for edge deletions.

(2) Novel maintenance algorithms. To efficiently handle bi-core maintenance with edge insertions, we introduce a novel structure called BD-Order, which offers a theoretical assurance regarding the scope of visited vertices. The BD-Order for a bipartite graph G comprises sets of vertex orders, detailing the sequences in which vertices are removed during bi-core decomposition (BD-Order may

---

[2]https://en.wikipedia.org/wiki/Wikipedia:Size_of_Wikipedia

differ for different bi-core decomposition algorithms). Specifically, it encompasses vertex orders corresponding to all possible α and β values, with each order representing the vertices within the respective bi-cores. For instance, in the bipartite graph illustrated in Figure 1 (excluding edge (u_4, v_2)), if $\alpha = 2$, the associated order could be $\{u_4, v_2, v_4, u_2, u_5, v_1, v_3, u_1, u_3\}$. This order signifies the removing sequence of all vertices during the computation of all $(2, \cdot)$-cores.

Building upon the BD-Order, we introduce two novel concepts: the *lack value* and the *support value*, which together provide a unified framework for efficiently managing bi-core updates. The lack value quantifies the number of additional neighbors required for a vertex to join a new bi-core, while the support value represents the number of neighbors a vertex retains within a specific bi-core. For edge insertions, we propose an efficient order-based algorithm that leverages the BD-Order to identify vertices potentially affected by the updates. The algorithm evaluates the lack values of these vertices, and determines whether their c-pairs, defined as integer pairs (α, β) corresponding to the bi-core they belong to, need updates. This approach reduces the number of visited vertices and minimizes the computation of neighbor information, enhancing overall efficiency. We also establish the near-bounded complexity of this algorithm through theoretical analysis. For edge deletions, we utilize the support value to develop a bounded algorithm for maintaining bi-cores. Unlike the insertion case, where the BD-Order significantly accelerates the updates, the deletion algorithm focuses on maintaining this structure. We analyze the algorithm's complexity and provide a theoretical guarantee of its efficiency.

(3) Experimental study. We extensively evaluate the efficiency of our proposed maintenance algorithms on both real and synthetic datasets. Experimental results reveal that for the edge insertion case, our algorithm is up to two orders of magnitude faster than the state-of-the-art approach. Moreover, our experiments show that the primary time consumption of the state-of-the-art algorithm lies in searching for the candidate set. In contrast, our proposed algorithm significantly reduces the time cost of this part, being up to 800× faster than the state-of-the-art approach in this part. Besides, for the edge deletion case, our algorithm is up to one order of magnitude faster, underscoring the efficacy of the support value we introduced in expediting the update process.

Outline. In Section 2, we introduce our research problem. Section 3 analyzes the boundedness of the problem studied and existing algorithms, while Section 4 discusses BD-Order used in our algorithms. Our bi-core maintenance algorithms for edge insertion and edge deletion are presented in Sections 5 and 6, respectively. Experimental results are reported in Section 7. We review related works in Section 8 and conclude in Section 9. For lack of space, all proofs in this paper are included in our full paper [47].

2 Problem Statement

Consider an undirected and unweighted bipartite graph $G = (U, V, E)$, where U and V denote two disjoint sets of vertices, and $E \subseteq U \times V$ is the set of edges between U and V in G. The numbers of vertices and edges of G are represented by n and m respectively, where $n = |U| + |V|$ and $m = |E|$. The set of neighbors of a vertex u in G is denoted as $N(u, G)$, and the degree of vertex u is denoted as $d(u, G) = |N(u, G)|$. Given two vertex sets $U' \subseteq U$ and $V' \subseteq V$, the subgraph of G induced by them is denoted as $G' = (U', V', E')$.

DEFINITION 1 $((\alpha, \beta)$-CORE [15]). *Given a bipartite graph $G = (U, V, E)$ and positive integers α and β, the (α, β)-core of G is the maximal subgraph with vertex sets $U' \subseteq U$ and $V' \subseteq V$, where each vertex in U' has at least α neighbors in V' and each vertex in V' has at least β neighbors in U'. That is, $\forall u \in U', d(u, G') \geq \alpha \wedge \forall v \in V', d(v, G') \geq \beta$.*

Table 1. Notations and meanings.

Notation	Meaning
$G=(U, V, E)$	A bipartite graph G with two disjoint vertex sets U and V and an edge set $E \subseteq U \times V$
n, m	The numbers of vertices and edges in G, respectively
$N(u, G)$	The set of neighbors of a vertex u in G
$N_k(W, G)$	The multiset of k-hop neighbors corresponding to the vertices in multiset W in G
$d(u, G)$	The degree of a vertex u in G
$G'=(U', V', E')$	The subgraph induced by $U' \subseteq U$ and $V' \subseteq V$
$A_{\alpha=i}(u, G)$	The anchored coreness of u regarding $\alpha = i$
$A_{\beta=j}(v, G)$	The anchored coreness of v regarding $\beta = j$
$G_{\alpha=i}, G_{\beta=j}$	The $(i, 0)$-core and $(0, j)$-core of G, respectively

We adopt a vertex-centric view to support efficient maintenance of bi-cores in dynamic bipartite graphs. Instead of maintaining entire bi-core subgraphs, we track for each vertex the set of bi-cores it belongs to, using the notion of a *c-pair*.

DEFINITION 2 (C-PAIR [30]). *Given a bipartite graph $G = (U, V, E)$, for a vertex $v \in U \cup V$, if there exists a (α, β)-core such that v is included in it, then we call (α, β) a* c-pair *of v.*

Each c-pair captures a specific bi-core membership of a vertex. By maintaining the c-pairs of all vertices, we can reconstruct all bi-cores without explicitly storing them. To summarize this membership more compactly, we define anchored coreness.

DEFINITION 3 (ANCHORED CORENESS). *For a vertex $u \in G$, and a fixed bi-core parameter $\alpha = i$ (resp. $\beta = j$), the anchored coreness $A_{\alpha=i}(u, G)$ (resp. $A_{\beta=j}(u, G)$) is the maximum β (resp. α) such that $u \in (i, \beta)$-core (resp. (α, j)-core).*

If no such (i, β)-core contains u, then we define the anchored coreness $A_{\alpha=i}(u, G) = 0$. Note that $\alpha = 0$ is disregarded due to its lack of practical relevance. Given a positive integer i, $G_{\alpha=i}$ denotes the maximal subgraph where all vertices belong to U have at least i neighbors, i.e., the $(i, 0)$-core of G. Similarly, $G_{\beta=j}$ is the $(0, j)$-core of G. Table 1 summarizes the frequently used notations.

The (α, β)-core exhibits two interesting properties: (1) an (α, β)-core is not necessarily connected; and (2) the (α, β)-cores have a partially nested relationship, as follows.

PROPERTY 1 ([26, 30]). *Given a bipartite graph G and two bi-cores, say (α_1, β_1)-core and (α_2, β_2)-core, if $\alpha_1 \geq \alpha_2$ and $\beta_1 \geq \beta_2$, then the (α_1, β_1)-core is a subgraph of the (α_2, β_2)-core.*

We now formally present the bi-core maintenance problem.

PROBLEM 1 ((α, β)-CORE MAINTENANCE [26, 30]). *Given a bipartite graph G and all its bi-cores, update these bi-cores after the insertion or deletion of an edge (u, v).*

Note that in line with previous works [26, 30], we mainly focus on the graph updates of edge insertion and edge deletion, since a vertex insertion and a vertex deletion are equivalent to a list of edge insertions and edge deletions, respectively.

3 Analysis of Problem and Algorithms

This section analyzes the boundedness of the bi-core maintenance problem and identifies the limitations of existing algorithms. To address these issues, we adopt the notion of near boundedness, a refined complexity measure tailored for unbounded cases.

3.1 Boundedness Analysis

Boundedness analysis is a common theoretical framework in incremental algorithms. We first present its key concepts and notations, then provide our theoretical analysis and results.

• **Notations.** Let Q be the query to decompose all the (α, β)-cores in a bipartite graph, with $Q(G)$ denoting the result of G's bi-core decomposition. D represents the bi-core decomposition algorithm, and M denotes the incremental bi-core maintenance algorithm. Given an update ΔG (e.g., an inserted or deleted edge), the updated query is $Q(G \oplus \Delta G)$, where $G \oplus \Delta G$ represents the updated graph. Let $D(G \oplus \Delta G)$ denote computing the query from scratch using algorithm D, and $M(G, \Delta G)$ represent the incremental computation with algorithm M. The change in the output is ΔR, so that $Q(G \oplus \Delta G) = Q(G) \oplus \Delta R$.

• **Concepts of boundedness.** The notion of *boundedness* [34] evaluates the effectiveness of an incremental algorithm using the metric CHANGED, defined as CHANGED $= \Delta G + \Delta R$ with $|$CHANGED$| = |\Delta G| + |\Delta R|$, which reflects a lower bound of work required to maintain the query.

DEFINITION 4 (BOUNDEDNESS [34, 49]). *An incremental algorithm is bounded if there is a polynomial function f and a positive integer k such that for any G and ΔG, the cost of the algorithm is bounded by $O(f(|N_k(\text{CHANGED}, G)|))$. Otherwise, it is unbounded.*

Similarly, an incremental problem is bounded if there exists a bounded incremental algorithm; otherwise, it is unbounded.

• **Our analysis.** To formalize our analysis, we adopt the computation model of *locally persistent* (LP) algorithms, which is widely used for analyzing incremental updates in graph algorithms [4, 16–18, 34, 49]. An LP algorithm maintains a storage block for each edge, containing pointers to adjacent edges and auxiliary status information. No global state is preserved across updates; all information is localized to edge-level storage. Existing bi-core maintenance algorithms rely only on local edge-level information and preserve no global state. Thus, they naturally fit the LP model, making it appropriate for our boundedness analysis. Updates proceed by traversing local pointers from the modified edge, with edge status updated dynamically along the path.

THEOREM 3.1. *Given a bipartite graph G and all its bi-cores, updating these bi-cores after inserting an edge (u, v) is unbounded under the model of locally persistent algorithms.*

PROOF SKETCH. *We prove this by contradiction. We begin by constructing a bipartite graph G with n vertices and defining two updates, each consisting of a single edge insertion, denoted as ΔG_1 and ΔG_2, which will be applied to G, and satisfy the following properties: for both ΔG_1 and ΔG_2, the corresponding ΔR is empty; after applying ΔG_1 and ΔG_2 sequentially to G, the $|\Delta R|$ is $\Omega(n)$.*

Assuming a bounded LP algorithm M exists, the workload for M to process either $G \oplus \Delta G_1$ or $G \oplus \Delta G_2$ would be $O(1)$. However, based on the properties of the constructed graph, we can prove that the total maintenance cost for any LP algorithm M to handle $G \oplus \Delta G_1$ and $G \oplus \Delta G_2$ separately is $\Omega(l)$, where l is a non-constant value. This leads to a contradiction with the assumption of a bounded algorithm. The complete proof can be found in our full paper [47].

THEOREM 3.2. *Given a bipartite graph G and all its bi-cores, updating these bi-cores after deleting an edge (u, v) is bounded.*

PROOF SKETCH. *During the bi-core maintenance for edge deletion, if the current vertex is to be removed from the original bi-core, it is added to CHANGED, leading to a visit to its neighbors. Otherwise, if it remains in the bi-core, neighbor visits are skipped. Thus, the maintenance time cost is $O(|$CHANGED$| + |N_1(\text{CHANGED}, G)|)$, making it bounded. The complete proof is in our full paper [47].*

Theorems 3.1 and 3.2 highlight the asymmetry of the bi-core maintenance problem: handling edge deletions is significantly simpler than handling edge insertions. This asymmetry distinguishes

bi-core maintenance from other incremental problems, such as incremental maximum cardinality matching in bipartite graphs [40], where both insertions and deletions are unbounded.

3.2 Analysis of Existing Algorithms

In the following, we examine how the above theoretical insights manifest in existing bi-core maintenance algorithms and identify their practical bottlenecks. Specifically, we analyze two representative methods [26, 30], both of which follow a similar high-level framework comprising three stages: 1) identifying the c-pairs to be analyzed, 2) computing the candidate set for each c-pair, and 3) updating the c-pairs of vertices. Algorithm 1 summarizes the detailed steps.

Algorithm 1: A framework for existing algorithms

input : G, c-pairs of all vertices, the updated edge (u, v)
output: Updated c-pairs of vertices in G
// Identify the c-pairs to be analyzed.
1 Identify the c-pair set P for potential updates;
// Compute the candidate set for each c-pair in P.
2 **foreach** $(\alpha, \beta) \in P$ **do**
3      $S, C \leftarrow \{u, v\}$;
4   **if** (u, v) *is an insertion edge* **then**
5          $(\alpha', \beta') \leftarrow$ the target c-pair;
6   **while** $S \neq \emptyset$ **do**
7          $w \leftarrow S.\text{pop}()$;
8   **foreach** $w' \in N(w, G)$ **do**
9   **if** (u, v) *is an insertion edge* **then**
10                  $S.\text{push}(w'), C.\text{insert}(w')$;
11   **if** $w' \notin (\alpha', \beta')$-*core* **then** $C.\text{erase}(w')$;
12   **else if** (α, β) *is no longer a c-pair of* w' **then**
13                  $S.\text{push}(w'), C.\text{insert}(w')$;
14      $(\alpha', \beta') \leftarrow$ the updated c-pair;
15   Update c-pairs of vertices in C with (α', β');

Given a bipartite graph G, the c-pairs of all vertices, and an edge (u, v) to be inserted or deleted, Algorithm 1 starts by identifying the c-pair set P for potential updates (line 1). For each c-pair (α, β) in P, a queue S stores vertices to be visited, and a set C represents the candidate set where the c-pairs of each vertex will be updated. Initially, vertices u and v are added to S and C (line 3). For edge insertion, the algorithm first obtains the target c-pair (α', β'), then iterates over each vertex w in S and checks if its neighbor w' belongs in the (α', β')-core; if not, w' is removed from C (lines 4-11). For edge deletion, the algorithm checks and adds vertices no longer in the bi-core of the current c-pair to C (lines 12-13). This process continues until S is empty, forming a BFS manner traversal. Finally, the updated c-pair is computed, and the c-pairs of the vertices in C are updated accordingly (lines 14-15).

Theorem 3.3 states the time complexity of the above framework.

THEOREM 3.3. *When handling an inserted edge (u, v), Algorithm 1 costs $O(m \cdot \max\{d(u, G), d(v, G)\})$ time. When handling a deleted edge (u, v), the time complexity of Algorithm 1 is $O(|\text{CHANGED}| + |N_1(\text{CHANGED}, G)|)$.*

As shown in Theorem 3.3, the time complexity for an edge insertion is $O(m \cdot \max\{d(u, G), d(v, G)\}) = \Omega(|\text{CHANGED}| \cdot \max\{d(u, G),$

$d(v, G)\}$), which indicates that the time cost is unbounded. In contrast, the time complexity for the edge deletion case is bounded. The removal-friendly nature of bi-core maintenance is evident in the ease of incrementalization for edge deletions, as vertices' c-pairs can be naturally computed using bi-core decomposition algorithms [26].

• **Limitations.** Existing algorithms focus on efficiently identifying the c-pairs to be analyzed. Liu et al. [26] proposed a maintenance algorithm to reduce the number of c-pairs that need to be updated. However, it still requires verifying a large number of c-pairs because the set P includes all c-pairs within a certain range of α and β. To address the above issue, Luo et al. [30] introduced the concept of bi-core number, which efficiently narrows the scope of P when handling edge insertions and deletions, significantly reducing computational redundancy and enhancing efficiency.

Table 2. Proportion of time cost of BFS process in state-of-the-art bi-core maintenance algorithm.

Datasets	Edge Insertion		Edge Deletion	
	BFS Process	Other	BFS Process	Other
AM	94.68%	5.32%	45.15%	54.85%
LS	96.42%	3.58%	87.18%	12.82%
DT	85.11%	14.89%	57.05%	42.95%
DBLP	91.27%	8.73%	52.27%	47.73%
ER	96.27%	3.73%	89.72%	10.28%
DE	93.46%	6.54%	85.81%	14.19%
DUI	98.39%	1.61%	78.78%	21.22%
LG	97.59%	2.41%	74.75%	25.25%

Overall, existing algorithms primarily focus on improving line 1 of Algorithm 1. However, the BFS-manner process (lines 6-13) within this framework still incurs considerable time overhead, impacting overall efficiency. To show the time cost occupied by the BFS process for handling edge insertion and edge deletion, we conduct an experiment using the state-of-the-art algorithm [30] on eight real-world datasets. The results shown in Table 2 indicate that the BFS process for identifying the candidate set C consumes a significant portion of time. Hence, there is substantial room for improving performance in searching for candidate sets during updates. These observations motivate the following question: *Is it possible to design algorithms for bi-core maintenance under edge insertions that are efficient in practice, even if the problem is theoretically unbounded?*

3.3 Toward Near Boundedness

To address the above question, we introduce a more refined notion called near boundedness, which provides a practical measure of complexity for unbounded yet structured problems. Algorithms that satisfy near boundedness typically achieve strong performance in practice and are therefore regarded as *practically efficient*, even if strict worst-case bounds do not hold.

• **Concepts of near boundedness.** In practice, |CHANGED| is often small. As a result, even for unbounded problems, some incremental algorithms remain practical by running in polynomial time to a measure comparable to |CHANGED|. To formalize this observation, Fan et al. [16, 17] proposed the concept of *relative boundedness* to evaluate the performance of incremental algorithms in unbounded settings. This concept applies to fully unbounded problems, where both insertions and deletions are unbounded, but does not extend to asymmetric problems such as bi-core maintenance. To address this, Zhang et al. [49] introduced the notion of *near boundedness*, tailored for asymmetric cases. We adopt this framework in our analysis. Before introducing near boundedness formally, we define the *affected part* (AFF), which serves as a complexity measure comparable to CHANGED. It

denotes a portion of the graph involved in recomputation and serves to characterize the theoretical cost of incremental algorithms in unbounded scenarios.

DEFINITION 5 (AFF [49]). *Given a graph G, a query Q, and an input update ΔG to G, AFF signifies the minimum difference between the data inspected by D when computing $Q(G)$ and the data inspected by any decomposition algorithm when computing $Q(G \oplus \Delta G)$.*

AFF captures the differences in executing the recompute algorithm. For unbounded problems, to evaluate the performance of the incremental algorithm, AFF provides a more accurate tool than CHANGED as it encompasses CHANGED, accurately reflects the necessary cost for maintaining the query result, and is only slightly larger than CHANGED in practice [49]. If M is an incremental algorithm with bounded time cost concerning |AFF|, then M is considered a near bounded incremental algorithm and thus is practically efficient. This leads to the following definition of near boundedness.

DEFINITION 6 (NEAR BOUNDEDNESS [49]). *An incremental graph algorithm M for Q is near bounded if there is a polynomial function f and a positive integer k such that for any G and ΔG, the cost of the algorithm is bounded by $O(f(|N_k(\text{AFF}, G)|))$.*

For bi-core maintenance, although edge insertions are unbounded, a near bounded algorithm may still exist. This motivates our design of a new maintenance framework that targets near boundedness by leveraging bi-core decomposition order, which will be introduced in the next section.

4 Bi-core Decomposition Order

Existing bi-core maintenance algorithms heavily rely on BFS-based local search to propagate updates. While effective in sparse regions, these approaches become inefficient in dense areas and lack support for precise complexity analysis. To address this, we propose a novel structure called the bi-core decomposition order (BD-Order), which records the sequence of vertex removals during the bi-core decomposition process. BD-Order is constructed naturally without additional overhead, enabling both efficient update propagation and theoretical analysis.

4.1 Structure of BD-Order

Since the bi-core model involves two parameters α and β, the BD-Order is defined along two dimensions. For a fixed $\alpha = i$, all $(i, \cdot)$-cores can be computed in a single pass by iteratively removing vertices. The vertices removed during this process form a sequence: $O_{\alpha=i} = \{w_1, w_2, \ldots, w_n\}$. If a vertex w_p is removed before w_q in this sequence, we denote this as $w_p \preceq_{\alpha=i} w_q$. Similarly, for a fixed $\beta = j$, the $(\cdot, j)$-core decomposition gives the sequence $O_{\beta=j}$. Varying α and β yields a total of 2δ sequences, where δ is the largest integer such that a (δ, δ)-core exists in the graph [26]. The BD-Order is formally defined as follows.

DEFINITION 7 (BD-ORDER). *In a bipartite graph G, the BD-Order of G is a set of vertex removal sequences:*

$$O_G = \{O_{\alpha=1}, O_{\alpha=2}, \cdots, O_{\alpha=\delta}, O_{\beta=1}, O_{\beta=2}, \cdots, O_{\beta=\delta}\},$$

where each $O_{\alpha=i}$ (resp. $O_{\beta=j}$) corresponds to the vertex removal sequence in the $(i, \cdot)$-core (resp. $(\cdot, j)$-core) decomposition process.

Table 3 shows an example of BD-Orders for the graph in Figure 1. The full BD-Order captures the entire bi-core decomposition process of G. In the subsequent discussion, each element of O_G (e.g., $O_{\alpha=i}$) is referred to as an order for clarity.

Implementation. We adopt the well-established *Order Data Structure* [6, 14] to maintain a total order of vertices with constant-time operations and linear space. This data structure supports the following three core primitives:

Table 3. The BD-Order for graph G.

Order	Vertices
$O_{\alpha=1}$	$v_2 \preceq v_4 \preceq v_3 \preceq u_5 \preceq v_1 \preceq u_1 \preceq u_2 \preceq u_3 \preceq u_4$
$O_{\alpha=2}$	$u_4 \preceq v_2 \preceq v_4 \preceq u_2 \preceq u_5 \preceq v_1 \preceq v_3 \preceq u_1 \preceq u_3$
$O_{\beta=1}$	$u_4 \preceq u_2 \preceq u_5 \preceq u_1 \preceq u_3 \preceq v_1 \preceq v_2 \preceq v_3 \preceq v_4$
$O_{\beta=2}$	$u_4 \preceq u_2 \preceq u_5 \preceq v_2 \preceq v_4 \preceq u_1 \preceq u_3 \preceq v_1 \preceq v_3$

- $\texttt{OPrec}(O_{\alpha=i}, x, y)$: Returns whether x precedes y in $O_{\alpha=i}$.
- $\texttt{OIns}(O_{\alpha=i}, x, y)$: Inserts element y after x in $O_{\alpha=i}$.
- $\texttt{ODel}(O_{\alpha=i}, x)$: Removes element x from $O_{\alpha=i}$.

Each element in O is assigned a label that encodes its position in the order. These labels allow $\texttt{OPrec}$ and $\texttt{ODel}$ to run in $O(1)$ time. The $\texttt{OIns}$ operation also takes $O(1)$ time when there is sufficient label space between x and its successor; otherwise, a localized relabeling is triggered, which still guarantees $O(1)$ amortized time.

4.2 Auxiliary Variables based on BD-Order

We define two variables for each vertex: *lack value* and *support value*, which are derived from the BD-Order and help identify affected vertices during updates. The lack value measures how many additional neighbors a vertex needs to qualify for inclusion in a new bi-core. The support value counts how many neighbors of a vertex remain within a specific bi-core, and thus determines whether it should stay in that core after an edge deletion.

DEFINITION 8 (LACK VALUE). *Given a bipartite graph G and integers i and j, the lack value of a vertex w in the order $O_{\alpha=i}$ is denoted as:*

$$L(i, w, O_{\alpha=i}) = \begin{cases} i - x, & \text{if } w \in U, \\ A_{\alpha=i}(w, G) + 1 - x, & \text{if } w \in V, \end{cases}$$

where $x = |\{w' \in N(w, G) \mid w \preceq_{\alpha=i} w'\}|$.
Similarly, the lack value in the order $O_{\beta=j}$ is denoted as:

$$L(i, w, O_{\beta=j}) = \begin{cases} A_{\beta=j}(w, G) + 1 - x, & \text{if } w \in U, \\ j - x, & \text{if } w \in V, \end{cases}$$

where $x = \left|\{w' \in N(w, G) \mid w \preceq_{\beta=j} w'\}\right|$.

DEFINITION 9 (SUPPORT VALUE). *Given a bipartite graph G and integers i and j, the support value of a vertex w in the order $O_{\alpha=i}$ is denoted as:*

$$S(i, w, O_{\alpha=i}) = |\{w' \in N(w, G) \mid A_{\alpha=i}(w, G) \leq A_{\alpha=i}(w', G)\}|.$$

Similarly, the support value in the order $O_{\beta=j}$ is denoted as:

$$S(i, w, O_{\beta=j}) = \left|\{w' \in N(w, G) \mid A_{\beta=j}(w, G) \leq A_{\beta=j}(w', G)\}\right|.$$

The lack value and support value play complementary roles in maintaining bi-cores under updates. The lack value determines whether a vertex qualifies for inclusion in a higher core after an insertion, while the support value is used to verify whether a vertex should remain in its current core after a deletion. Both quantities are computed to the BD-Order and help to significantly reduce redundant local computations during update propagation.

EXAMPLE 1. *In Figure 1, for graph G (excluding edge (u_4, v_2)), we have $\delta = 2$. After bi-core decomposition, the BD-Order is presented in Table 3. Focusing on $\alpha = 2$, where $O_{\alpha=2} = \{u_4, v_2, v_4, u_2, u_5, v_1, v_3, u_1, u_3\}$, the corresponding lack values for these vertices are {1, 1, 1, 1, 1, 1, 1, 2, 2}, as per Definition 8; the support values for these vertices are {1, 2, 2, 2, 2, 3, 3, 3, 3}, as per Definition 9. Lack values and support values for vertices in other orders of O_G can be computed in the same way.*

4.3 Formulation of AFF based on BD-Order

Now we formalize the AFF regarding BD-Order, which quantifies the minimum cost required to extend the bi-core decomposition algorithm to a maintenance algorithm.

DEFINITION 10 (AFF VIA BD-ORDER). *Let O_G and $O'_{G+\Delta G}$ denote the BD-Orders before and after an update ΔG applied to G. Then:*

$$\text{AFF} = \min_{valid\ O'} \text{diff}(O_G, O'_{G+\Delta G})$$

where

$$\text{diff}(O_G, O'_{G \oplus \Delta G}) = \{\cup_{i=1}^{\delta} [\text{diff}(O_{\alpha=i}, O'_{\alpha=i}) \cup \text{diff}(O_{\beta=i}, O'_{\beta=i})]\}.$$

Here, $\text{diff}(O_{\alpha=i}, O'_{\alpha=i})$ includes vertices whose positions in $O'_{\alpha=i}$ have shifted backward relative to their positions in the original order $O_{\alpha=i}$ and it can be formally represented as follows:

$$\text{diff}(O_{\alpha=i}, O'_{\alpha=i}) = \{w \in G | \exists w' \in G, w \preceq_{\alpha=i} w' \wedge w' \preceq'_{\alpha=i} w\},$$

$\text{diff}(O_{\beta=i}, O'_{\beta=i})$ can be represented in the same way. Definition 10 formalizes the theoretical cost required to maintain the consistency of the decomposition after changes. The AFF includes both CHANGED (input/output difference) and the order-drift induced by edge insertions or deletions. Since $O'_{G \oplus \Delta G}$ is not unique, AFF for BD-Order is defined as the minimum difference among all possible $\text{diff}(O_G, O'_{G \oplus \Delta G})$. This formulation provides a principled lower bound on the cost required to maintain the bi-core decomposition after an update.

EXAMPLE 2. *In the graph of Figure 1, (u_4, v_2) is the insertion edge. Consider $\alpha = 2$, the initial order $O_{\alpha=2}$ is $\{u_4, v_2, v_4, u_2, u_5, v_1, v_3, u_1, u_3\}$. After insertion, the new order $O'_{\alpha=2}$ becomes $\{v_4, u_5, v_3, u_1, u_4, v_2, u_2, v_1, u_3\}$. The differences between the initial and new orders indicate variations in O_G and $O'_{G \oplus \Delta G}$. Note that $O'_{G \oplus \Delta G}$ is non-unique due to the variable vertex peeling sequence in the bi-core decomposition. As defined above, AFF denotes the minimum discrepancy between O_G and any possible $O'_{G \oplus \Delta G}$.*

5 Order-based Edge Insertion

In this section, we present an order-based algorithm for maintaining bi-cores under edge insertions. Given an inserted edge (u, v), let G^+ denote the updated graph. Without loss of generality, assume $u \in U$ and $v \in V$. The algorithm proceeds in two key steps: (1) identifying the candidate set of vertices whose c-pairs require updates, and (2) updating their c-pairs while preserving the BD-Order for efficient future updates.

5.1 Analytical Scope of the Candidate Set

Section 2 demonstrates that the BFS process for identifying the candidate set (Algorithm 1, lines 6-13) consumes a significant portion of time in the existing edge insertion algorithms. To reduce this cost, we use the BD-Order to localize updates. Since the operations on $O_{\alpha=i}$ and $O_{\beta=j}$ are symmetric, we focus on $O_{\alpha=i}$ in the following discussion without loss of generality. We begin by presenting the following lemma.

LEMMA 5.1. *Let G be a bipartite graph, $O_{\alpha=i}$ an associated order, and (u,v) an inserted edge with $u \preceq_{\alpha=i} v$. Then, a c-pair update may occur for vertices in the original $(i, \cdot)$-cores **only if** $L(i, u, O_{\alpha=i}) = 1$. In this case, u is the first vertex added to the candidate set.*

EXAMPLE 3. *Consider the edge insertion (u_4, v_2) in Figure 1, under $\alpha = 2$. Since $u_4 \preceq_{\alpha=2} v_2$ and u_4 is first determined to enter the $(2, 2)$-core, we subsequently examine whether it qualifies for the $(2, 3)$-core. At this point, $L(2, u_4, O_{\alpha=2}) = 1$, satisfying the necessary condition for the update to occur. After further verification, u_4 is confirmed to belong to the $(2, 3)$-core in G^+, and its lack value $L(2, u_4, O_{\alpha=2})$ is updated to 2. Consequently, even if u_4 connects to additional vertices, it cannot be added to the $(2, 4)$-core.*

Theorem 5.1 establishes a necessary condition for triggering updates after an edge insertion. Specifically, inserting (u,v) decreases $L(i, u, O_{\alpha=i})$ by at most one, as it introduces only a single new neighbor for u. If u still lacks sufficient neighbors appearing after it in $O_{\alpha=i}$ to support an update ($L(i, u, O_{\alpha=i}) > 0$ after it is decreased), no vertices in the original $(i, \cdot)$-cores will be added to the candidate set. Otherwise, $L(i, u, O_{\alpha=i}) = 0$, u is added to the candidate set, triggering further updates. Once u's c-pair is updated, its lack value is also updated to reflect the change. Building on this, the following lemma further restricts the update scope by identifying the vertices that may be impacted.

LEMMA 5.2. *Given a bipartite graph G, an order $O_{\alpha=i}$, and an insertion edge (u,v) with $u \preceq_{\alpha=i} v$, a vertex $w \in G$ can belong to the candidate set **only if** the following conditions hold simultaneously: (1) $A_{\alpha=i}(u, G) \leq A_{\alpha=i}(w, G) \leq A_{\alpha=i}(v, G)$, and (2) there exists a path $w_0, w_1, \ldots, w_t$ such that $w_0 = u$, $w_t = w$, and for each $0 \leq p < t$, $(w_p, w_{p+1}) \in E$ with $w_p \preceq_{\alpha=i} w_{p+1}$.*

EXAMPLE 4. *Consider the graph G in Figure 1 with the inserted edge (u_4, v_2) and order $O_{\alpha=2}$. We have $u_4 \preceq_{\alpha=2} v_2$, $A_{\alpha=2}(u_4, G) = 0$, and $A_{\alpha=2}(v_2, G) = 2$. According to Theorem 5.2, Figure 2 illustrates possible paths starting from u_4, where an arrow $w \rightarrow w'$ denotes $(w, w') \in E$ and $w \preceq_{\alpha=2} w'$. Thus, besides u_4 and v_2, vertices v_1, u_1, u_3, and u_2 qualify as potential candidates. In contrast, vertices u_5, v_3, and v_4 are excluded since no such path exists connecting them to u_4.*
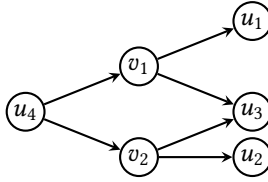

Fig. 2. Paths where the vertices satisfy Lemma 5.2 in $O_{\alpha=2}$.

5.2 Order-based Candidate Set Identification

Leveraging the lemmas in Section 5.1, we can precisely determine the scope of affected vertices following the insertion of edge (u,v) into G. We then present an order-based algorithm for identifying the candidate set, which consists of vertices whose c-pairs require updates.

Given a graph G, a c-pair (i, j), and an insertion edge (u,v) with $u \preceq_{\alpha=i} v$, Algorithm 2 outlines the procedure for identifying all vertices in the candidate set C that will be newly included in the (i, j)-core of G^+. The algorithm adopts a two-phase strategy: **candidate expansion** and **candidate verification**. In the first phase, candidates are optimistically selected based on lack values, under the assumption that all subsequent neighbors in the order provide valid support. Since this assumption may not always hold, the second phase verifies and refines the candidate set by propagating removals, ensuring that only valid vertices are retained.

Algorithm 2: Order-Based-Ins($G, (u, v), (i, j)$)

---

input : $G, (u, v)$, target c-pair (i, j) ▷ assume $u \preceq_{\alpha=i} v$
output: the candidate set C

1   $S, C, T \leftarrow \emptyset$;
2   **if** $i \leq j$ **then**
     // Phase 1: Candidate expansion
3   **if** *the condition of Theorem 5.1 is satisfied* **then**
4       $C.\text{insert}(u);$ $S.\text{push}(u);$
5       $L(i, u, O_{\alpha=i}) \leftarrow L(i, u, O_{\alpha=i}) - 1;$
6   **while** $S \neq \emptyset$ **do**
7       $w \leftarrow S.\text{pop}();$
8   **for** $w' \in N(w, G^+)$ **do**
9   **if** w' *satisfies the conditions of Theorem 5.2* **then**
10          $L(i, w', O_{\alpha=i}) \leftarrow L(i, w', O_{\alpha=i}) - 1;$
11   **if** $L(i, w', O_{\alpha=i}) = 0$ **then**
12   **if** $w' \in T$ **then** $T.\text{erase}(w');$
13            $S.\text{push}(w');$ $C.\text{insert}(w');$
14   **else if** $L(i, w', O_{\alpha=i}) > 0$ **then** $T.\text{insert}(w');$

     // Phase 2: Candidate verification
15   **while** $T \neq \emptyset$ **do**
16       $w \leftarrow T.\text{pop}();$ ▷ Freeze T into a queue
17   **for** $w' \in N(w, G^+)$ **do**
18   **if** $w' \in C \wedge \text{OPrec}(O_{\alpha=i}, w', w)$ **then**
19           $L(i, w', O_{\alpha=i}) \leftarrow L(i, w', O_{\alpha=i}) + 1;$
20   **if** $L(i, w', O_{\alpha=i}) > 0$ **then**
21             $C.\text{erase}(w');$ $T.\text{push}(w');$

22   **else**
23   lines 3-21 by swapping i with j and α with β.

Specifically, the algorithm initializes three data structures: a queue S for storing vertices to be visited during BFS traversal, a set C representing the candidate set, and a set T containing vertices that follow u in the order and are not yet in the (i, j)-core of G^+ but may become candidates through propagation (line 1). If $i \leq j$, the algorithm proceeds using the order $O_{\alpha=i}$ (line 2). If $i > j$, the algorithm performs analogous operations by interchanging i with j and α with β, following the same logic as lines 3–21 (lines 22–23).

In the candidate expansion phase, the algorithm begins with vertex u. If the condition in Theorem 5.1 is satisfied, u is inserted into both C and S (lines 3–4). Following the insertion of edge (u, v), $L(i, u, O_{\alpha=i})$ is decremented by one, reflecting the addition of a new neighbor (line 5). The algorithm iterates over each vertex w in S (lines 6–14). For every neighbor w' of w in G^+ that satisfies the condition in Theorem 5.2, it decrements $L(i, w', O_{\alpha=i})$ by one (lines 8–10). If $L(i, w', O_{\alpha=i}) = 0$, w' is removed from T (if present) and added to both C and S (lines 11–13). Otherwise, if $L(i, w', O_{\alpha=i}) > 0$, it is added to T for further verification in the next phase (line 14).

In the candidate verification phase, the algorithm processes each vertex in T to exclude vertices in C that do not satisfy the update conditions (lines 15–21). For each vertex $w \in T$, the algorithm checks whether its neighbors w' in G^+ remain in the candidate set C (lines 16-17). For every

neighbor $w' \in C$ such that $w' \prec_{\alpha=i} w$, the value $L(i, w', O_{\alpha=i})$ is incremented by one (lines 18–19). If the updated value satisfies $L(i, w', O_{\alpha=i}) > 0$, then w' is removed from C and added to T (lines 20–21). This process repeats until the set T becomes empty (line 15). As noted, Algorithm 2 uses $\mathrm{OPrec}(O_{\alpha=i}, w, w')$ to check if w precedes w' in $O_{\alpha=i}$, which can be done in $O(1)$ time.

EXAMPLE 5. *Consider the graph G in Figure 1, where the edge (u_4, v_2) is to be inserted. Since u_4 belongs to the $(2, 2)$-core, we verify whether it qualifies for the $(2, 3)$-core. At this point, $L(2, u_4, O_{\alpha=2}) = 1$. According to Theorem 5.1, u_4 is added to the candidate set C and the queue S. Next, $L(2, v_1, O_{\alpha=2})$ and $L(2, v_2, O_{\alpha=2})$ are each decremented by one, since the following conditions hold: (1) $v_1, v_2 \in N(u_4, G^+)$; (2) $A_{\alpha=2}(v_1, G) = A_{\alpha=2}(v_2, G) = 2$; and (3) $u_4 \preceq_{\alpha=2} v_1$, $u_4 \preceq_{\alpha=2} v_2$. After the decrements, both v_1 and v_2 have lack values of zero, and are thus added to C and S. The algorithm then continues to process the neighbors of v_1, namely u_1 and u_3, following lines 8–9 of Algorithm 2. Since $L(2, u_1, O_{\alpha=2}) = L(2, u_3, O_{\alpha=2}) = 1 > 0$ after decrease, both vertices are added to set T. Once the queue S is empty, the intermediate sets are $C = \{u_4, v_1, v_2, u_2, u_3\}$ and $T = \{u_1\}$, with $L(2, u_1, O_{\alpha=2}) = 1$.*

In the verification phase, $u_1 \in T$ is processed. Among its neighbors, only v_1 satisfies the condition in line 18 of Algorithm 2, leading to an increment of $L(2, v_1, O_{\alpha=2})$ back to zero. Since v_1 remains in C, no changes occur. The algorithm then terminates with the final candidate set $C = \{u_4, v_1, v_2, u_2, u_3\}$.

5.3 BD-Order Maintenance

To ensure correct computation and maintain consistency with lack value updates, we preserve the BD-Order throughout the process. Consider the c-pair (i, j), the original order $O_{\alpha=i}$, and the insertion edge (u, v) with $u \preceq_{\alpha=i} v$. Let $O_{\alpha=i}(j)$ denote the subsequence of vertices in $O_{\alpha=i}$ satisfying the anchored coreness $A_{\alpha=i}(\cdot, G) = j$. These subsequences $O_{\alpha=i}(1), O_{\alpha=i}(2), \ldots$ are arranged such that $O_{\alpha=i}(p) \preceq_{\alpha=i} O_{\alpha=i}(q)$ whenever $p < q$. To obtain the updated order $O'_{\alpha=i}$ for G^+, we replace the subsequences $O_{\alpha=i}(j-1)$ and $O_{\alpha=i}(j)$ in the original order with their updated counterparts $O'_{\alpha=i}(j-1)$ and $O'_{\alpha=i}(j)$, without modifying the rest of the order. In $O'_{\alpha=i}(j-1)$, vertices are partitioned into two groups: those that shift backward relative to their original positions (i.e., vertices added to T in line 14 of Algorithm 2), and those that remain unchanged. For $O'_{\alpha=i}(j)$, vertices from the final candidate set are inserted at the beginning of the subsequence, preserving their relative order. Note that u is a special case. Since $A_{\alpha=i}(u, G)$ may differ from $j-1$, u is first removed from its original group $O_{\alpha=i}(A_{\alpha=i}(u, G))$ and then inserted at the beginning of $O'_{\alpha=i}(j)$.

(a) Original order $O_{\alpha=2}$

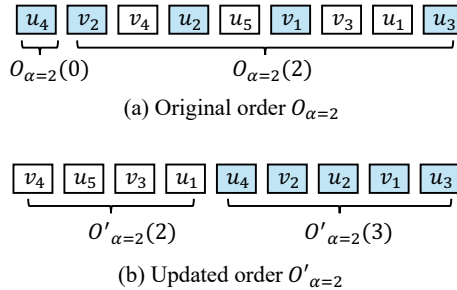(b) Updated order $O'_{\alpha=2}$

Fig. 3. The idea of order maintenance with edge insertion.

EXAMPLE 6. *Reconsider Example 5, we illustrate how the order $O_{\alpha=2}$ is maintained in Figure 3. Initially, the group $O_{\alpha=2}(2)$ consists of the sequence $\{v_2, v_4, u_2, u_5, v_1, v_3, u_1, u_3\}$. After the insertion of edge (u_4, v_2), the degree of u_4 increases from 1 to 2, thereby qualifying it for inclusion in $O_{\alpha=2}(2)$. According to the update strategy, u_4 is inserted at the beginning of $O_{\alpha=2}(2)$. During Algorithm 2, vertex u_1 remains in T after phase 1, and upon completing phase 2, the vertices in candidate set*

are u_4, v_2, u_2, v_1, u_3. Thus u_1 *should be shifted backward in* $O'_{\alpha=2}(2)$, *while* u_4, v_2, u_2, v_1, u_3 *should be removed. The resulting revised subsequence is* $O'_{\alpha=2}(2) = \{v_4, u_5, v_3, u_1\}$. *After placing the candidate vertices at the beginning of* $O'_{\alpha=2}(3)$, *we obtain* $O'_{\alpha=2}(3) = \{u_4, v_2, u_2, v_1, u_3\}$. *Combining all updated parts, the final order becomes* $O'_{\alpha=2} = \{v_4, u_5, v_3, u_1, u_4, v_2, u_2, v_1, u_3\}$.

5.4 Overall Edge Insertion Algorithm

Building on the BD-Order-based candidate set identification, we propose an efficient algorithm for handling edge insertions. The algorithm leverages the framework described in Algorithm 1, where the c-pair set P is to be analyzed in line 1. Instead of executing lines 6–13 of Algorithm 1, we invoke Algorithm 2 to compute the candidate set. Since the primary computational cost of Algorithm 1 arises from lines 6–13, this substitution significantly enhances overall performance. We subsequently provide a comprehensive analysis of the algorithm, including its correctness, time complexity, and boundedness.

Correctness. To prove the correctness of the proposed edge insertion algorithm, we first introduce the following lemma for the correctness of BD-Order maintenance.

LEMMA 5.3. *Given a graph G, an inserted edge (u, v), the target c-pair (i, j), and the order $O_{\alpha=i}$, for any vertex w belongs to the candidate set, the updated $L(i, w, O_{\alpha=i})$ is not less than its value before the update.*

Based on this result, we state the following theorem regarding the correctness of the entire algorithm.

THEOREM 5.4. *Given a bipartite graph G and an insertion edge (u, v), our proposed edge insertion algorithm updates c-pairs of all the vertices and the BD-Order correctly.*

Time complexity and boundedness. Given a graph G and an insertion edge (u, v), the time complexity of the overall algorithm is described as follows.

THEOREM 5.5. *Let (u, v) be an inserted edge in G, the time complexity of the proposed edge insertion algorithm is $O(|N_1(\text{AFF}, G^+)| \cdot |N_2(\text{AFF}, G^+)|)$, where $\text{AFF} = \text{diff}(O_G, O'_{G \oplus \Delta G})$. The algorithm is near bounded to the bi-core decomposition algorithm [26], as its cost can be expressed as a polynomial function of $|\text{AFF}|$.*

This result highlights that the algorithm only accesses a small portion of the vertices compared to existing approaches, explaining its superior empirical performance.

6 Bounded Edge Deletion

In this section, we present an efficient bi-core maintenance algorithm for handling edge deletions. Let (u, v) be the deleted edge, and denote the updated graph as G^-. Without loss of generality, we assume $u \in U$ and $v \in V$. As with edge insertion, the procedure consists of two main steps: identifying the candidate set of vertices whose c-pairs require updates after the deletion, and subsequently maintaining the BD-Order to facilitate future updates. However, unlike insertion, edge deletion is a bounded problem. As a result, the BD-Order plays a less critical role in the candidate identification phase, though it is still preserved to ensure consistency across updates.

6.1 Candidate Set Identification

We focus on $O_{\alpha=i}$ in the following discussion, as operations on $O_{\beta=j}$ are symmetric. We begin with the necessary and sufficient conditions for the occurrence of updates.

LEMMA 6.1. *Let G be a bipartite graph, $O_{\alpha=i}$ an associated order, and (u, v) a deleted edge. Then, the c-pair update may occur for vertices in the original $(i, \cdot)$-cores if and only if at least one of the following conditions is satisfied:*

(1) *If $A_{\alpha=i}(u, G) \leq A_{\alpha=i}(v, G)$, and $S(i, u, O_{\alpha=i}) = i$, then u is added into the candidate set.*
(2) *If $A_{\alpha=i}(u, G) \geq A_{\alpha=i}(v, G)$, and $S(i, v, O_{\alpha=i}) = A_{\alpha=i}(v, G)$, then v is added into the candidate set.*

EXAMPLE 7. *In the graph G shown in Figure 1 (contains the dashed line here), let (u_4, v_2) be the edge to be deleted. Considering $\alpha = 2$, we find that $A_{\alpha=2}(u_4, G) = A_{\alpha=2}(v_2, G) = 3$. Initially, $S(2, u_4, O_{\alpha=2}) = 2$ and $S(2, v_2, O_{\alpha=2}) = 3$, indicating that the update will occur and both u and v are added into the candidate set.*

LEMMA 6.2. *Given a bipartite graph G, an order $O_{\alpha=i}$, and a deletion edge (u, v) with $u \preceq_{\alpha=i} v$, a vertex $w \in G$ can belong to the candidate set **only if** the following conditions hold simultaneously: (1) $A_{\alpha=i}(w, G) = A_{\alpha=i}(u, G)$, and (2) there exists a path $w_0, w_1, \ldots, w_t$ such that $w_t = w$, $w_0 = u$ or $w_0 = v$ (when $A_{\alpha=i}(u, G) = A_{\alpha=i}(v, G)$), and for every $0 \leq p < t$, $(w_p, w_{p+1}) \in E$ and $A_{\alpha=i}(w_p, G) = A_{\alpha=i}(u, G)$.*

Lemma 6.2 characterizes the set of vertices that may be affected by edge deletions, thereby enabling a more precise identification of the update scope.

Algorithm 3: Bounded-Del$(G, (u, v), (i, j))$

---

input : G, (u, v), and original c-pair (i, j)
output : the candidate set C

1   $S, C \leftarrow \emptyset$;
2   **if** $i \leq j$ **then**
3   **if** *condition (1) of Theorem 6.1 is satisfied* **then**
4   C.insert(u); S.push(u);
5       $S(i, u, O_{\alpha=i}) \leftarrow S(i, u, O_{\alpha=i}) - 1$;
6   **if** *condition (2) of Theorem 6.1 is satisfied* **then**
7   C.insert(v); S.push(v);
8       $S(i, v, O_{\alpha=i}) \leftarrow S(i, v, O_{\alpha=i}) - 1$;
9   **while** $S \neq \emptyset$ **do**
10       $w \leftarrow S$.pop();
11   **for** $w' \in N(w, G^-)$ **do**
12   **if** w' *satisfies the conditions of Theorem 6.2* **then**
13           $S(i, w', O_{\alpha=i}) \leftarrow S(i, w', O_{\alpha=i}) - 1$;
14   **if** $S(i, w', O_{\alpha=i}) = i - 1 \wedge w' \in U$ **then**
15   C.insert(w'); S.push(w');
16   **if** $S(i, w', O_{\alpha=i}) = j - 1 \wedge w' \in V$ **then**
17   C.insert(w'); S.push(w');

18   **else**
19   Lines 3-17 by swapping i with j, α with β, and U with V.

Based on these lemmas, we propose Algorithm 3 to identify the candidate set C, which consists of vertices to be removed from certain bi-cores. Given a graph G, an edge (u, v) to be deleted, and an original c-pair (i, j), the algorithm initializes a queue S and a set C to track the vertices to be processed and the candidates for removal from the (i, j)-core in G, respectively (line 1). If $i \leq j$, the

algorithm begins with order $O_{\alpha=i}$ (line 2). It uses Theorem 6.1 to check if u and v should be added to S and C, and updates their support values (lines 3-8). Next, the algorithm processes each vertex in S (lines 9-10). For a neighbor w' of the current visited vertex w, if it satisfy Theorem 6.2, the algorithm decreases its support values by one, then adds it to S and C if $w' \in U$ and $S(i, w', O_{\alpha=i}) = i - 1$ or $w' \in V$ and $S(i, w', O_{\alpha=i}) = j - 1$ (lines 11-17). If $i > j$, the operations are similar to lines 3-17 by swapping i with j, α with β, and U with V (lines 18-19).

EXAMPLE 8. *Given graph G in Figure 1, where the edge (u_4, v_2) is to be deleted, we consider the original c-pair $(2, 3)$, and use the order $O_{\alpha=2}$. We observe that $A_{\alpha=2}(u_4, G) = A_{\alpha=2}(v_2, G) = 3$, $S(2, u_4, O_{\alpha=2}) = 2$, and $S(2, v_2, O_{\alpha=2}) = 3$, leading to the addition of u_4 and v_2 into S and C, along with updates to their support values. Subsequently, for neighbor v_1 of u_4, its support value decreases to 2, and it is added to S and C. Further processing of vertices in S yields termination of Algorithm 3 with $C = \{u_4, v_1, v_2, u_2, u_3\}$.*

6.2 BD-Order Maintenance

Compared to order maintenance under edge insertion, the update of BD-Order in the case of edge deletion is more straightforward. Specifically, consider the c-pair (i, j), the original order $O_{\alpha=i}$, and the edge (u, v) to be deleted with $u \preceq_{\alpha=i} v$. For each vertex w in the candidate set with $w \neq u$, we remove w from $O_{\alpha=i}(j)$ and append it to the end of $O'_{\alpha=i}(j - 1)$. Vertex u is a special case, as its updated value $A_{\alpha=i}(u, G^-)$ may differ from $j - 1$. After removing u from $O_{\alpha=i}(j)$, it is appended to the end of $O'_{\alpha=i}(A_{\alpha=i}(u, G^-))$.

EXAMPLE 9. *Reconsider Example 8, we maintain the order $O_{\alpha=2}$ as follows. Initially, we have $O_{\alpha=2}(2) = \{v_4, v_5, v_3, u_1\}$ and $O_{\alpha=2}(3) = \{v_1, v_2, u_2, u_3, u_4\}$. Algorithm 3 finds that the candidate set is $C = \{u_4, v_2, v_1, u_2, u_3\}$. So, we remove these vertices from $O_{\alpha=2}(3)$ and add them into the end of $O'_{\alpha=2}(2)$ with the order resulting from the candidate finding process. Finally, $O'_{\alpha=2} = \{v_4, v_5, v_3, u_1, u_4, v_2, v_1, u_2, u_3\}$.*

6.3 Overall Edge Deletion Algorithm

Building upon the methods proposed for candidate set identification and BD-Order maintenance, we can outline the efficient algorithm for handling edge deletions. First, we leverage the framework outlined in Algorithm 1, to obtain the c-pairs for analysis. Then, we utilize Algorithm 3 to identify newly deleted vertices from the specified (i, j)-core in G^- and compute the candidate set. Next, we conduct a comprehensive analysis of the algorithm, encompassing its correctness and time complexity.

Correctness. To prove the correctness of the proposed edge deletion algorithm, we have Theorem 6.3 for the overall algorithm.

THEOREM 6.3. *Given a bipartite graph G and a deletion edge (u, v), our proposed edge deletion algorithm updates c-pairs of all the vertices and the BD-Order correctly.*

Time complexity. The time complexity of the overall edge deletion algorithm can be described by the following theorem.

THEOREM 6.4. *Given one deleted edge (u, v) in graph G, the time complexity of the proposed edge deletion algorithm is $O(|\text{CHANGED}| + |N_1(\text{CHANGED}, G)|)$. This also shows that the proposed edge deletion algorithm is bounded.*

7 Experiments

7.1 Setup

Datasets. We utilize 12 large bipartite graphs, comprising ten real graphs sourced from KONECT[3], and two synthetic graphs. The synthetic graphs are constructed using a bipartite network model [8], featuring a power-law (PL) graph with a power-law degree distribution achieved through random edge addition, and a uniform degree (UD) graph where edges are uniformly distributed. Table 4 reports the statistics of each graph, where $|U|$ and $|V|$ are numbers of vertices on the two sides of graphs respectively, and $|E|$ is the number of edges.

Table 4. Bipartite graphs used in the experiments.

Graphs	Abbr.	Category	$|U|$	$|V|$	$|E|$
edit-kwwiktionary	EK	Interaction	187	1.06K	3.37K
Bonanza	BA	Authorship	7.92K	1.97K	36.54K
Actor movies	AM	Movie	0.13M	0.38M	1.47M
Last.fm songs	LS	Song	0.99K	1.10M	4.41M
Discogs artist–style	DT	Feature	1.62M	0.38K	5.74M
dblp	DBLP	Authorship	1.95M	5.62M	12.28M
Epinions	ER	Rating	0.12M	0.76M	13.67M
Wikipedia edits (de)	DE	Authorship	1.03M	5.91M	55.23M
Delicious user–item	DUI	Folksonomy	0.83M	33.78M	101.80M
LiveJournal	LG	Affiliation	3.20M	7.49M	112.31M
Power law	PL	Random	5M	5M	1B
Uniform degree	UD	Random	5M	5M	1B

Algorithms. We test the following maintenance algorithms:

- `Re-compute` [26]: the state-of-the-art bi-core decomposition algorithm;
- `BII`[*] [26]: the index-based bi-core maintenance algorithm to handle edge insertion;
- `BIR`[*] [26]: the index-based bi-core maintenance algorithm to handle edge deletion;
- `BNI` [30]: the state-of-the-art bi-core maintenance algorithm to handle edge insertion;
- `BNR` [30]: the state-of-the-art bi-core maintenance algorithm to handle edge deletion;
- `OI`: our proposed order-based bi-core maintenance algorithm for handling edge insertion;
- `BD`: our proposed bounded bi-core maintenance algorithm for handling edge deletion.

Note that our order-based bi-core maintenance algorithm is orthogonal to existing methods and thus can be integrated with any of them. In this work, we combine it with the method from [26], as bi-core numbers [30] alone cannot capture changes in relative ordering. Consequently, c-pairs pruned by bi-core numbers still require verification, which limits the effectiveness of pruning. We also assess some additional experimental results for extending these algorithms to batch-processing scenarios. Due to space limitations, the pertinent experimental results are included in our full paper [47].

Experimental settings. To evaluate the efficiency of bi-core maintenance algorithms above, for each graph, we select r edges to simulate the edge insertion and edge deletion process, where $r \in \{5K, 10K, 15K, 20K, 25K\} \cup \{1\% |E|, 2\% |E|, 3\% |E|, 4\% |E|, 5\% |E|\}$. For graphs with real timestamps, edges are selected in temporal order; otherwise, edges are selected uniformly at random. When testing edge insertions, we first remove the selected r edges from the original graphs and then insert them back into the graph sequentially. When testing edge deletions, we directly delete these r edges one by one from the original graphs. The default setting of r is 5,000, following prior studies [26, 30]. Note that for the EK dataset, where the total number of edges is smaller than 5,000,
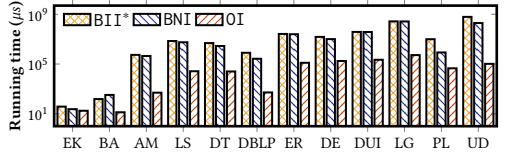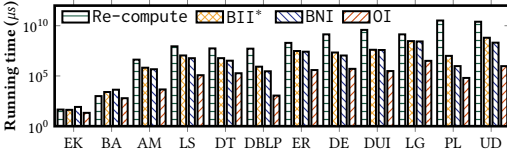
---

[3]http://konect.cc/networks/

Fig. 4. Efficiency of an edge insertion on all datasets. Fig. 5. The average time to find the candidate sets after edge insertions.
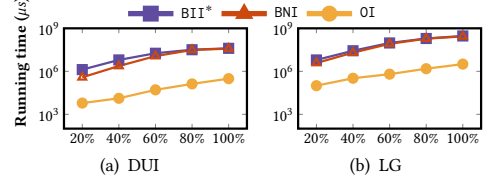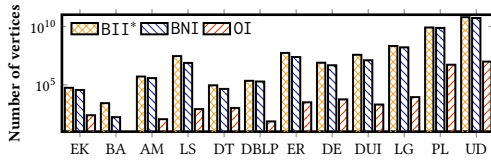

Fig. 6. The average number of visited vertices after Fig. 7. Scalability test for handling edge insertions. edge insertions.
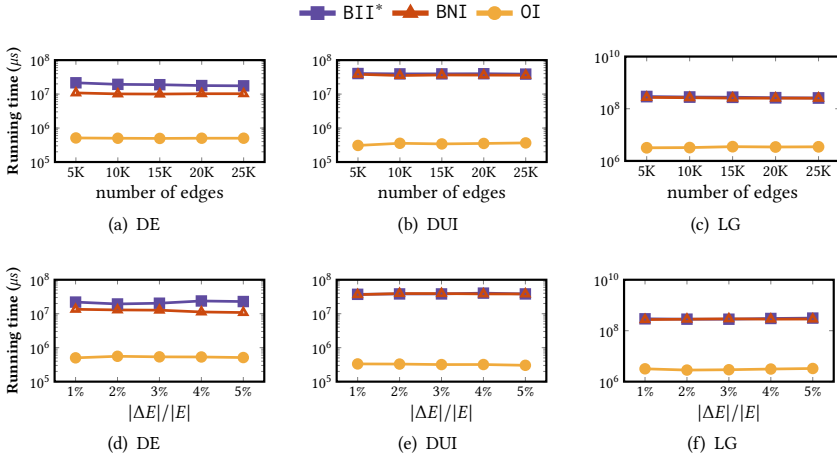

Fig. 8. Effect of the number and ratio of inserted edges (average time per edge).

we instead select 1-5% of the edges as update edges, with 1% used by default. All the algorithms above are implemented in C++ and compiled with the g++ compiler at -O3 optimization level. All the experiments are run on a Linux machine with an Intel Xeon 2.40GHz CPU and 512GB RAM. In the following reported time cost, each data point is the average result of r edge insertions or deletions.

7.2 Efficiency of Edge Insertions

• **Overall efficiency results.** Figure 4 shows the average time cost of an edge insertion on different datasets. Across all datasets, Re-compute method exhibits the poorest performance, as it redundantly recalculates all bi-cores in the graph, even for unmodified parts. Our algorithms show significant performance improvements compared to baseline methods. For example, on the DBLP dataset, OI processes edge insertions in 1.114 ms, compared to 52007.5 ms for Re-compute, 837.081 ms for BII* and 291.204 ms for BNI. This indicates that our algorithm is up to two orders
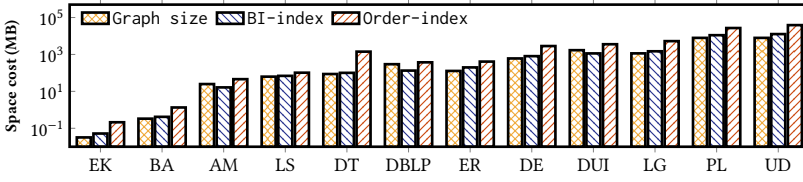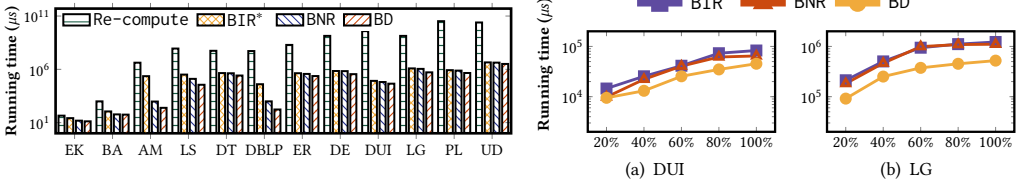
Fig. 9. The size of graphs and indexes.

Fig. 10. Efficiency of an edge deletion on all datasets. Fig. 11. Scalability test for handling edge deletions.

of magnitude faster than the state-of-the-art algorithm. The primary reason for this improvement is that our proposed algorithm can accurately identify vertices impacted by insertions, allowing updates to be completed by accessing smaller subgraphs. In contrast, BNI necessitates analyzing additional vertices and their neighbor information, while BII* further requires analyzing additional c-pairs, leading to the need for larger subgraph accesses.

• **Time cost of candidate set identification.** Figure 5 depicts the time required by BII*, BNI, and OI to identify candidate sets while inserting 5,000 edges per edge. Our algorithm OI demonstrates a notable reduction in time compared to the state-of-the-art algorithm BNI. This reduction is significant because the majority of the SOTA algorithm's time is dedicated to searching for candidate sets. Additionally, Figure 6 presents the average number of vertices visited by BII*, BNI, and OI per edge during 5,000 edge insertions. These results highlight the BD-Order's significant role in reducing the number of visited vertices, thus enhancing search efficiency.

• **Effect of the number and ratio of inserted edges.** Figure 19 illustrates the effect of both the number and ratio of inserted edges on processing time, where $r \in 5K, 10K, 15K, 20K, 25K$ and $r \in \{1\%\,|E|, 2\%\,|E|, 3\%\,|E|, 4\%\,|E|, 5\%\,|E|\}$. Since Re-compute always decomposes the entire graph, its running time remains stable and is therefore omitted here. We report results on the three largest real-world datasets, as other datasets exhibit consistent trends and are omitted for brevity. The consistent behavior observed in both absolute-count and ratio-based settings demonstrates the stability and scalability of our method under varying insertion sizes.

• **Scalability test.** To test the scalability, we randomly selected 20%, 40%, 60%, 80%, and 100% of edges from each graph, and then obtained five induced subgraphs by these edges. We only show the results on DUI and LG when inserting 5,000 edges in Figure 7 since the trends are similar on all other datasets. The running time of all algorithms increases with the number of edges, while our method consistently outperforms BII* and BNI, demonstrating good scalability on large graphs.

• **Space Cost of Indexes.** Figure 9 shows the memory usage for the graphs, the BI-index from [26], and the Order-index produced by our methods across all datasets. In some cases, the BI-index requires more storage than the graph itself, while in others, it requires less. Overall, the sizes of the BI-index and the graph are relatively similar. The Order-index, however, generally requires about three times more storage than the BI-index due to the need to store additional information for the BD-Order. In the case of the DT dataset, our approach requires approximately ten times more memory, attributed to the dataset's sparsity. Despite this higher memory usage, the substantial performance improvements justify the additional cost.
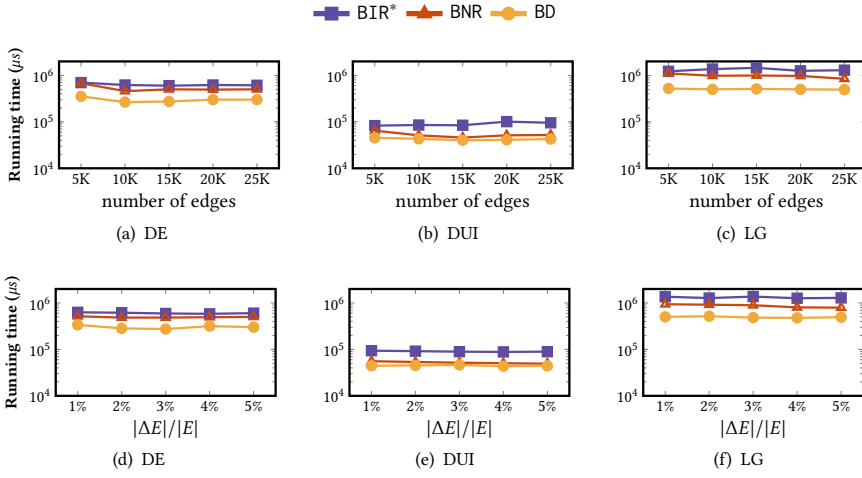
Fig. 12. Effect of the number and ratio of deleted edges (average time per edge).

7.3 Efficiency of Edge Deletions

• **Overall efficiency results.** Figure 10 presents the average edge deletion time across various datasets. Similar to edge insertions, Re-compute performs poorly on all datasets. However, during edge deletions, BIR*, BNR, and BD significantly outperform Re-compute, exhibiting greater superiority compared to edge insertions. This is because bi-core maintenance with edge deletion is already bounded, limiting the vertices incremental algorithms need to visit. Consequently, additional bounding via BD-Order is unnecessary. Nevertheless, our proposed support value still enhances performance by providing supplementary information. For example, on DBLP, our algorithm can handle an edge deletion in 0.171 ms, while Re-compute, BIR*, and BNR need 52007.5 ms, 40.170 ms, and 0.989 ms respectively. Compared to the state-of-the-art algorithm, the improvement offered by our proposed deletion algorithm is relatively slight, since BD additionally needs to maintain the BD-Order. However, it is still faster than BIR* and BNR across all datasets.

• **Effect of the number and ratio of deleted edges.** Figure 12 shows the effect of the number and ratio of deleted edges on processing time, where $r \in \{5K, 10K, 15K, 20K, 25K\}$ and $r \in \{1\%\,|E|, 2\%\,|E|, 3\%\,|E|, 4\%\,|E|, 5\%\,|E|\}$. We only present the results on the three largest real-world graphs, because the trends are similar on all other datasets. Note that we also omit the running time of Re-compute here because it is stable. Generally, the findings indicate that with the number of edges increasing, there is no substantial rise in the average time it takes to delete each edge. This demonstrates the stability of our proposed algorithm under varying deletion sizes.

• **Scalability test.** To evaluate the scalability, we experimented by randomly selecting 20%, 40%, 60%, 80%, and 100% of the edges from each graph, and then obtained five induced subgraphs by these edges. We only conducted experiments on datasets DUI and LG when deleting 5,000 edges, because the trends are similar on all other datasets. As illustrated in Figure 11, the execution time for all algorithms increases with the addition of edge numbers. However, our algorithm consistently outperforms BIR* and BNR in all cases. Therefore, we can conclude that our maintenance algorithm for edge deletion achieves better scalability in practice.
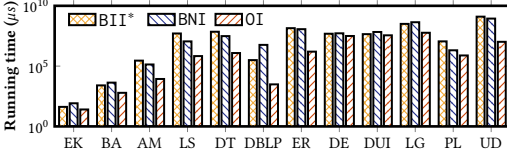
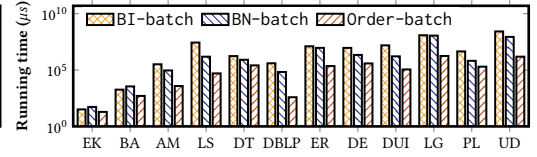Fig. 13. Stress test (average time per edge).

Fig. 14. Efficiency of batch update algorithms on all datasets.

7.4 Additional evaluations

• **Stress test.** In addition to temporal and uniform edge selection, we further include a worst-case stress test, where the 5,000 edges with the highest endpoint-degree sums are inserted. We conducted this stress test only for edge insertions, as insertions are generally more challenging than deletions. We report the average processing time across all datasets in Figure 13. The results reveal that the vertex degree distribution has a substantial impact on algorithmic performance. When inserted edges connect high-degree vertices on both sides, the processing time of all maintenance algorithms increases noticeably. On datasets such as DE, LG, and PL, the performance gap among maintenance algorithms narrows, as these cases closely approximate worst-case scenarios. Nevertheless, our algorithm consistently outperforms existing baselines, achieving up to two orders of magnitude speedups. This demonstrates that our approach remains efficient even when updates are concentrated in structurally challenging regions of the graph.
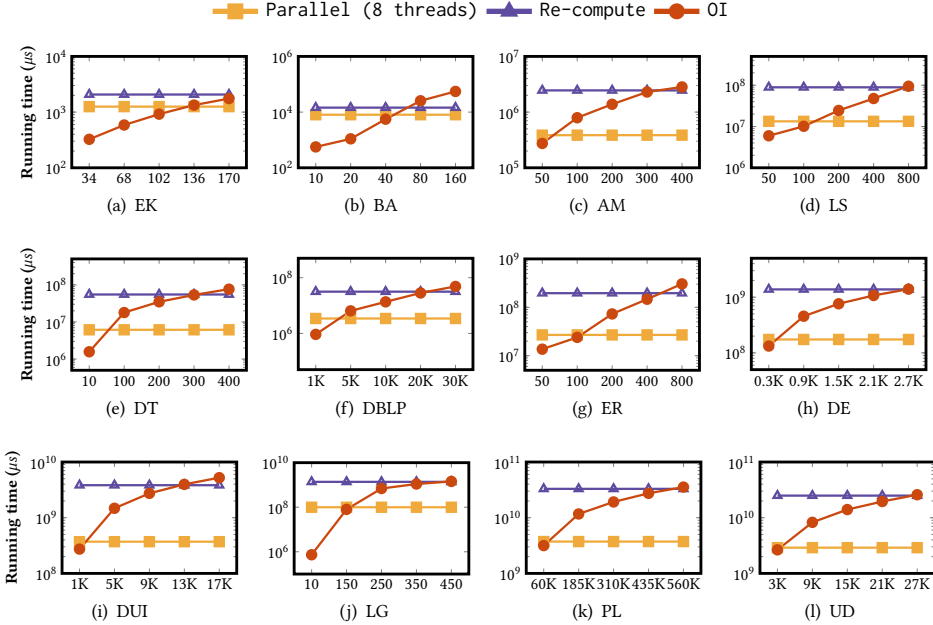

Fig. 15. Total time of maintenance vs. Re-compute.

• **Efficiency of batch updates.** Our bi-core maintenance algorithm can be easily adapted for batch updating by following the batch updating framework in [26, 30]. Specifically, we first handle the edges that are both inserted and deleted by removing them from the edge set to be updated. Next, we process the remaining inserted and deleted edges separately. For newly inserted edges, we first determine the c-pairs to be processed for all the endpoints and then collectively

update them using our proposed methods. These operations are similarly applied to deal with edge deletions. To evaluate the efficiency of the batch updates, we selected 5000 edges from each dataset, randomly designating them for insertion or deletion. Based on the results depicted in Figure 14, our `Order-batch` algorithm significantly outperforms the methods `BI-batch` and `BN-batch`, by achieving up to two orders of magnitude in average processing time per edge.

• **Maintenance vs. re-computation, crossover analysis.** To determine the range where our approach outperforms re-computation from scratch, we compare incremental maintenance with full re-computation under continuous update sequences and identify the crossover point where re-computation becomes comparable to maintenance. For timestamped datasets, updates follow temporal order to simulate real-world graph evolution; for datasets without timestamps, insertions are sampled uniformly at random. We also include a state-of-the-art parallel static baseline [22], denoted as `Parallel` and executed with 8 threads, to enable a more comprehensive comparison. As shown in Fig. 15, the crossover points vary significantly across datasets. This variation is primarily determined by the degree distribution of vertices. In datasets such as LG, where the inserted edges tend to connect high-degree vertices on both sides, the maintenance process becomes more time-consuming, leading to a smaller crossover threshold. In contrast, for smaller datasets such as EK and BA, the static decomposition itself is already fast, resulting in a relatively low crossover point as well.

8 Related Works

Cohesive subgraphs maintenance in unipartite graphs. Identifying cohesive subgraphs is crucial in graph analytics, with common types including k-core [37], k-truss [11], and k-clique [12]. Despite the availability of efficient algorithms for computing these subgraphs, updating them after edge insertions or deletions is computationally intensive. This has led to significant interest in dynamic graph maintenance algorithms, resulting in numerous methods for efficiently maintaining k-cores [2, 24, 35, 36, 45, 50], k-trusses [21, 39, 46, 49], and k-cliques [13, 38] after graph updates.

k-core is the most widely used cohesive subgraph and is closely related to the (α, β)-core in bipartite graphs. k-core maintenance algorithms typically update the core numbers of vertices to maintain the k-cores. Li et al. [24] and Sariyuce et al. [35] independently noted that the set of affected vertices remains connected after edge insertions or deletions. Leveraging on this observation, [35] proposed an algorithm with linear complexity relative to the size of affected vertices. Conversely, [24] proposed an algorithm with quadratic complexity. Zhang et al. [50] developed a core maintenance algorithm that preserves the k-order between any two vertices, enhancing computational efficiency by effectively identifying the vertices requiring updates. Liu et al. [27] introduced the first efficient parallel batch-dynamic algorithm for $(2 + \epsilon)$-approximate k-core decomposition, employing a novel parallel level data structure to achieve $O(|\mathcal{B}| \log^2 n)$ amortized work and $O(\log^2 n \log \log n)$ depth on an graph with n vertices and a batch $\mathcal{B}$ of updates. Guo et al. [20] observed that all existing parallel core maintenance methods rely on the sequential traversal algorithm, which limits parallelism when vertices share the same core numbers. To overcome this, they introduce a parallel algorithm based on the more efficient k-order method, which supports parallelism regardless of core number distribution. However, the k-core maintenance approaches cannot be directly applied to bi-core maintenance due to differences in subgraph structures.

Core maintenance in bipartite graphs. Some researchers have studied the maintenance of bi-cores in dynamic bipartite graphs [26, 30]. Liu et al. [26] utilized a two-level structure to index all bi-cores and developed an index maintenance algorithm for the (α, β)-core in dynamic bipartite graphs. Luo et al. [30] introduced bi-core numbers for vertices in bipartite graphs and analyzed how edge insertions and deletions affect these numbers, devising corresponding bi-core maintenance algorithms.

Although existing algorithms have achieved notable progress, they still suffer from redundant computations when traversing potentially affected vertices through BFS. In addition, the absence of boundedness analysis on the number of visited vertices leaves no theoretical guarantee on how vertex visits relate to the number of edge insertions or deletions. Hence, there remains considerable room for improving the efficiency of bi-core maintenance.

9 Conclusion

In this study, we investigate the bi-core maintenance problem, delving into its theoretical complexities under edge insertions and deletions. We prove that while bi-core maintenance remains bounded for edge deletions, it becomes unbounded with edge insertions, highlighting the intricate challenges involved. To tackle this issue, we introduce BD-Order, a novel structure, and utilize it to devise a near bounded edge insertion algorithm for efficient bi-core maintenance. Additionally, we present a complementary bounded edge deletion algorithm. Our experimental evaluations on both real-world and synthetic large-scale graphs validate the efficacy of our proposed algorithms. In our future work, we will focus on exploring efficient parallel algorithms for bi-core maintenance in large bipartite graphs, as well as exploring the maintenance of other cohesive subgraphs in bipartite graphs.

Acknowledgments

References

[1] Adel Ahmed, Vladimir Batagelj, Xiaoyan Fu, Seok-Hee Hong, Damian Merrick, and Andrej Mrvar. 2007. Visualisation and analysis of the internet movie database. In *2007 6th International Asia-Pacific Symposium on Visualization*. IEEE, 17–24.

[2] Hidayet Aksu, Mustafa Canim, Yuan-Chi Chang, Ibrahim Korpeoglu, and Özgür Ulusoy. 2014. Distributed *k*-Core View Materialization and Maintenance for Large Dynamic Graphs. *IEEE Transactions on Knowledge and Data Engineering* 26, 10 (2014), 2439–2452.

[3] Mohammad Allahbakhsh, Aleksandar Ignjatovic, Boualem Benatallah, Seyed-Mehdi-Reza Beheshti, Elisa Bertino, and Norman Foo. 2013. Collusion detection in online rating systems. In *Web Technologies and Applications: 15th Asia-Pacific Web Conference, APWeb 2013, Sydney, Australia, April 4-6, 2013. Proceedings 15*. Springer, 196–207.

[4] Bowen Alpern, Roger Hoover, Barry K Rosen, Peter F Sweeney, and F Kenneth Zadeck. 1990. Incremental evaluation of computational circuits. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*. 32–42.

[5] Wen Bai, Yadi Chen, Di Wu, Zhichuan Huang, Yipeng Zhou, and Chuan Xu. 2022. Generalized core maintenance of dynamic bipartite graphs. *Data Mining and Knowledge Discovery* (2022), 1–31.

[6] Michael A Bender, Richard Cole, Erik D Demaine, Martin Farach-Colton, and Jack Zito. 2002. Two simplified algorithms for maintaining order in a list. In *European Symposium on Algorithms*. Springer, 152–164.

[7] Alex Beutel, Wanhong Xu, Venkatesan Guruswami, Christopher Palow, and Christos Faloutsos. 2013. Copycatch: stopping group attacks by spotting lockstep behavior in social networks. In *Proceedings of the 22nd international conference on World Wide Web*. 119–130.

[8] Etienne Birmelé. 2009. A scale-free graph model based on bipartite graphs. *Discrete Applied Mathematics* 157, 10 (2009), 2267–2284.

[9] Monika Cerinšek and Vladimir Batagelj. 2015. Generalized two-mode cores. *Social Networks* 42 (2015), 80–87.

[10] Hongxu Chen, Hongzhi Yin, Tong Chen, Weiqing Wang, Xue Li, and Xia Hu. 2020. Social boosted recommendation with folded bipartite network embedding. *IEEE Transactions on Knowledge and Data Engineering* 34, 2 (2020), 914–926.

[11] Jonathan Cohen. 2008. Trusses: Cohesive subgraphs for social network analysis. *National security agency technical report* 16, 3.1 (2008), 1–29.

[12] Maximilien Danisch, Oana Balalau, and Mauro Sozio. 2018. Listing k-cliques in sparse real-world graphs. In *Proceedings of the 2018 World Wide Web Conference*. 589–598.

[13] Apurba Das, Michael Svendsen, and Srikanta Tirthapura. 2019. Incremental maintenance of maximal cliques in a dynamic graph. *The VLDB Journal* 28 (2019), 351–375.

[14] Paul Dietz and Daniel Sleator. 1987. Two algorithms for maintaining order in a list. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. 365–372.

[15] Danhao Ding, Hui Li, Zhipeng Huang, and Nikos Mamoulis. 2017. Efficient fault-tolerant group recommendation using alpha-beta-core. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. 2047–2050.

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

[17] Wenfei Fan and Chao Tian. 2022. Incremental graph computations: Doable and undoable. *ACM Transactions on Database Systems (TODS)* 47, 2 (2022), 1–44.

[18] Wenfei Fan, Xin Wang, and Yinghui Wu. 2013. Incremental graph pattern matching. *ACM Transactions on Database Systems (TODS)* 38, 3 (2013), 1–47.

[19] Stephan Gunnemann, Emmanuel Muller, Sebastian Raubach, and Thomas Seidl. 2011. Flexible fault tolerant subspace clustering for data with missing values. In *2011 IEEE 11th International Conference on Data Mining*. IEEE, 231–240.

[20] Bin Guo and Emil Sekerinski. 2023. Parallel order-based core maintenance in dynamic graphs. In *Proceedings of the 52nd International Conference on Parallel Processing*. 122–131.

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

[22] Yihao Huang, Claire Wang, Jessica Shi, and Julian Shun. 2023. Efficient algorithms for parallel bi-core decomposition. In *2023 Symposium on Algorithmic Principles of Computer Systems (APOCS)*. SIAM, 17–32.

[23] Mehdi Kaytoue, Sergei O Kuznetsov, Amedeo Napoli, and Sébastien Duplessis. 2011. Mining gene expression data with pattern structures in formal concept analysis. *Information Sciences* 181, 10 (2011), 1989–2001.

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

[25] Boge Liu, Long Yuan, Xuemin Lin, Lu Qin, Wenjie Zhang, and Jingren Zhou. 2019. Efficient (α, β)-core computation: An index-based approach. In *The World Wide Web Conference*. 1130–1141.

[26] Boge Liu, Long Yuan, Xuemin Lin, Lu Qin, Wenjie Zhang, and Jingren Zhou. 2020. Efficient (α, β)-core computation in bipartite graphs. *The VLDB Journal* 29, 5 (2020), 1075–1099.

[27] Quanquan C Liu, Jessica Shi, Shangdi Yu, Laxman Dhulipala, and Julian Shun. 2022. Parallel batch-dynamic algorithms for k-core decomposition and related graph problems. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*. 191–204.

[28] Xiaowen Liu, Jinyan Li, and Lusheng Wang. 2008. Modeling protein interacting groups by quasi-bicliques: complexity, algorithm, and application. *IEEE/ACM transactions on computational biology and bioinformatics* 7, 2 (2008), 354–364.

[29] Wensheng Luo, Kenli Li, Xu Zhou, Yunjun Gao, and Keqin Li. 2022. Maximum Biplex Search over Bipartite Graphs. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 898–910.

[30] 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.

[31] Bingqing Lyu, Lu Qin, Xuemin Lin, Ying Zhang, Zhengping Qian, and Jingren Zhou. 2020. Maximum biclique search at billion scale. *Proceedings of the VLDB Endowment* (2020).

[32] Ziyi Ma, Yuling Liu, Yikun Hu, Jianye Yang, Chubo Liu, and Huadong Dai. 2022. Efficient maintenance for maximal bicliques in bipartite graph streams. *World Wide Web* 25, 2 (2022), 857–877.

[33] Eirini Ntoutsi, Kostas Stefanidis, Katharina Rausch, and Hans-Peter Kriegel. 2014. " Strength Lies in Differences" Diversifying Friends for Recommendations through Subspace Clustering. In *CIKM*. 729–738.

[34] Ganesan Ramalingam and Thomas Reps. 1996. On the computational complexity of dynamic graph problems. *Theoretical Computer Science* 158, 1-2 (1996), 233–277.

[35] Ahmet Erdem Saríyüce, Buğra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V Çatalyürek. 2013. Streaming algorithms for k-core decomposition. *Proceedings of the VLDB Endowment* 6, 6 (2013), 433–444.

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

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

[38] Shengli Sun, Yimo Wang, Weilong Liao, and Wei Wang. 2017. Mining maximal cliques on dynamic graphs efficiently by local strategies. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 115–118.

[39] Zitan Sun, Xin Huang, Qing Liu, and Jianliang Xu. 2023. Efficient star-based truss maintenance on dynamic graphs. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–26.

[40] Ismail H Toroslu and Göktürk Üçoluk. 2007. Incremental assignment problem. *Information Sciences* 177, 6 (2007), 1523–1529.

[41] Kai Wang, Yiheng Hu, Xuemin Lin, Wenjie Zhang, Lu Qin, and Ying Zhang. 2021. A Cohesive Structure Based Bipartite Graph Analytics System. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 4799–4803.

[42] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. 2020. Efficient bitruss decomposition for large-scale bipartite graphs. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 661–672.

[43] Kai Wang, Wenjie Zhang, Xuemin Lin, Ying Zhang, and Shunyang Li. 2022. Discovering hierarchy of bipartite graphs with cohesive subgraphs. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2291–2305.

[44] Kai Wang, Wenjie Zhang, Xuemin Lin, Ying Zhang, Lu Qin, and Yuting Zhang. 2021. Efficient and effective community search on large-scale bipartite graphs. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 85–96.

[45] Dong Wen, Lu Qin, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2016. I/O efficient core graph decomposition at web scale. In *2016 IEEE 32nd international conference on data engineering (ICDE)*. IEEE, 133–144.

[46] Tianyang Xu, Zhao Lu, and Yuanyuan Zhu. 2022. Efficient triangle-connected truss community search in dynamic graphs. *Proceedings of the VLDB Endowment* 16, 3 (2022), 519–531.

[47] Qiaoyuan Yang, Wensheng Luo, Yixiang Fang, and Yuanyuan Zeng. 2026. Order-based Algorithms for Efficient Core Maintenance in Large Bipartite Graphs (full paper). (2026). https://github.com/QuanQuan0/Bicore-Maintenance-Order-Full-Paper/blob/main/full_paper.pdf

[48] Yun Zhang, Charles A Phillips, Gary L Rogers, Erich J Baker, Elissa J Chesler, and Michael A Langston. 2014. On finding bicliques in bipartite graphs: a novel algorithm and its application to the integration of diverse biological data types. *BMC bioinformatics* 15 (2014), 1–18.

[49] Yikai Zhang and Jeffrey Xu Yu. 2019. Unboundedness and efficiency of truss maintenance in evolving graphs. In *Proceedings of the 2019 International Conference on Management of Data*. 1024–1041.

[50] Yikai Zhang, Jeffrey Xu Yu, Ying Zhang, and Lu Qin. 2017. A fast order-based approach for core maintenance. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 337–348.

A Proofs of lemmas and theorems

A.1 Proof of Theorem 3.1

PROOF. In the bipartite graph G illustrated in Figure 16 (excluding the two dotted edges (u_1, v_1) and (u_i, v_j)), there are i vertices in set U and j vertices in set V. Let ΔG_1 and ΔG_2 represent the insertion of edges $e_1 = (u_1, v_1)$ and $e_2 = (u_i, v_j)$, respectively. We define graphs $G_1 = G \oplus \Delta G_1$, $G_2 = G \oplus \Delta G_2$, and $G_3 = G_1 \oplus \Delta G_2$. Consider the query for the $(2, 2)$-core during bi-core decomposition, denoted as $Q_{(2,2)}$. We observe that $Q_{(2,2)}(G)$, representing the $(2, 2)$-core of graph G, contains no vertices. This is because vertices are sequentially removed during the decomposition, such as $u_1, v_1, v_2, u_2, \ldots$. The $(2, 2)$-cores of G_1 and G_2 remain empty for the same reason. Thus, $Q_{(2,2)}(G) = Q_{(2,2)}(G_1) = Q_{(2,2)}(G_2)$. However, $Q_{(2,2)}(G)$ and $Q_{(2,2)}(G_3)$ differ significantly, with $Q_{(2,2)}(G_3)$ containing all vertices in the graph.

We prove the theorem by contradiction. Assume there exists a bounded, locally persistent algorithm M for bi-core maintenance under edge insertion. Let $V_M(G, \Delta G)$ denote the sequence of vertices visited by M when updating G with ΔG. If such an algorithm exists, running $M(G, \Delta G_1)$ and $M(G, \Delta G_2)$ should take constant time, as only a single edge is inserted in each case. Consequently, the outputs must be identical, implying both $V_M(G, \Delta G_1)$ and $V_M(G, \Delta G_2)$ are of constant size, leading to $|V_M(G, \Delta G_1)| + |V_M(G, \Delta G_2)| = O(1)$.

Now, consider $G \oplus \Delta G_2$ and $G_1 \oplus \Delta G_2$. The executions of M on $G \oplus \Delta G_2$ and $G_1 \oplus \Delta G_2$ must differ because $Q_{(2,2)}(G_3)$ and $Q_{(2,2)}(G_2)$ are different. Let $V_M(G, \Delta G_2)$ and $V_M(G_1, \Delta G_2)$ represent the vertices visited in these executions. Since both executions start at (u_i, v_j) but result in different traces, there must be at least one vertex w where the information differs, and a path from (u_i, v_j) to w exists in $V_M(G, \Delta G_2)$. This difference is due to the distinction between G and G_1, suggesting that edge (u_1, v_1) alters the information of vertex w in G. Hence, a path exists from (u_1, v_1) to w in $V_M(G, \Delta G_1)$. Therefore, there is a path from (u_1, v_1) to (u_i, v_j) through w with a length of $\Omega(i + j)$. This contradicts the assumption that $M(G, \Delta G_1)$ and $M(G, \Delta G_2)$ can run constantly, proving that M cannot be bounded. □
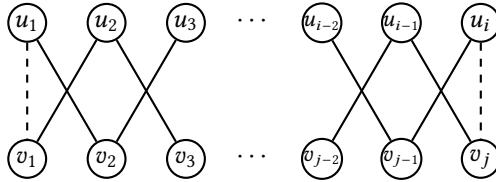

Fig. 16. Graph for proving unboundedness.

A.2 Proof of Theorem 3.2

PROOF. Let M represent a locally persistent bi-core maintenance algorithm for edge deletion. It begins by examining the (α, β)-core starting from endpoints of the deleted edge (u, v). If the current vertex is to be removed from the (α, β)-core, it's added to CHANGED, leading to a visit to its neighbors. Otherwise, if it remains in the (α, β)-core, neighbor visits are skipped. This process follows a BFS strategy. The time complexity is $O(M(G, \Delta G)) = O(|N_1(\text{CHANGED}, G)| + |\text{CHANGED}|)$. Thus, M is a bounded algorithm and the bi-core maintenance is a bounded problem for edge deletions. □

A.3 Proof of Lemma 5.1

PROOF. If $u \preceq_{\alpha=i} v$, the insertion of (u, v) introduces a new neighbor for u positioned after u in the order $O_{\alpha=i}$, resulting in a decrease of $L(i, u, O_{\alpha=i})$ by one. If $L(i, u, O_{\alpha=i})$ remains greater than 0

after this decrease, it indicates that u still lacks sufficient neighbors for inclusion in the new core, and no further update occurs. □

A.4 Proof of Lemma 5.2

PROOF. For condition (1), suppose the vertex w is in the candidate set with $A_{\alpha=i}(w, G) < A_{\alpha=i}(u, G)$. The contradiction arises because w being in the candidate set implies it gains new neighbors in the $(i, A_{\alpha=i}(w, G) + 1)$-core after the edge insertion. However, since $A_{\alpha=i}(w, G) < A_{\alpha=i}(u, G)$, both u and v are already in the $(i, A_{\alpha=i}(w, G) + 1)$-core originally, leading to no new neighbors for any vertex in that core. Now, suppose w in the candidate set with $A_{\alpha=i}(w, G) > A_{\alpha=i}(v, G)$, this implies that neither u nor v are part of the original $(i, A_{\alpha=i}(w, G))$-core. However, after inserting the edge (u, v) into graph G, updates to the corresponding c-pairs only affect vertices within bi-cores that originally contain u or v. Consequently, vertex w cannot be included in the candidate set.

For condition (2), the values of $L(i, w, O_{\alpha=i})$ may decrease due to the updates in u's c-pairs and $u \preceq_{\alpha=i} w$. These updates initiate with u and propagate locally. Therefore, of there exists a path $w_0, w_1, \cdots, w_t$ such that $w_0 = u$, $w_t = w$, $(w_p, w_{p+1}) \in E$, $w_p \preceq_{\alpha=i} w_{p+1}$ for each $0 \le p < t$, then the decrease in u's lack value can propagate to w's lack value locally. As a result, w may be included in the candidate set. Conversely, if condition (2) is not satisfied, w must not belong to the candidate set. □

A.5 Proof of Lemma 5.3

PROOF. Suppose w is a vertex in the candidate set. The positional relationship between w and its neighbor w' in the updated order $O'_{\alpha=i}$ after edge insertion depends on the comparison of their respective $A_{\alpha=i}(\cdot, G)$ values: (1) If $A_{\alpha=i}(w', G) \ge A_{\alpha=i}(w, G) + 1$, w''s position in $O'_{\alpha=i}$ remains after w; (2) If $A_{\alpha=i}(w', G) = A_{\alpha=i}(w, G)$ and w' is also in the candidate set, their positional relationship remains unchanged; (3) If $A_{\alpha=i}(w', G) = A_{\alpha=i}(w, G)$ and w' is not in the candidate set, w''s position in $O'_{\alpha=i}$ will be ahead of w after edge insertion, possibly increasing $L(i, w, O_{\alpha=i})$; (4) If $A_{\alpha=i}(w', G) < A_{\alpha=i}(w, G)$, w''s position is always ahead of w before and after edge insertion. In summary, for any vertex w in the candidate set, the updated $L(i, w, O_{\alpha=i})$ is not less than its value before the update, as demonstrated. □

A.6 Proof of Theorem 5.4

PROOF. In our edge insertion algorithm, we leverage the framework shown in Algorithm 1 to determine the original c-pair set for potential updates and calculate the target c-pairs. We then examine each vertex with suitable c-pairs, considering their adjacency to vertices in the candidate set based on BD-Order information. Notably, the induced subgraph of the candidate set remains connected. Consequently, our insertion algorithm ensures no potential vertices of the new bi-core are overlooked. Moreover, the algorithm correctly maintains BD-Order and the lack value of each vertex, as confirmed by Lemma 5.3 post-update. Thus, our algorithm accurately updates c-pairs for all vertices and preserves BD-Order integrity. □

A.7 Proof of Theorem 5.5

PROOF. Given a graph G and an insertion edge (u, v), consider a target c-pair (α, β) and an order $O_{\alpha=i}$ for an integer i with $u \preceq_{\alpha=i} v$. Let $\text{AFF}_{(\alpha, \beta)}$ represent the subset of AFF corresponding to the given target c-pair (α, β).

To calculate the time complexity of Algorithm 2, we divide the analysis into two parts. First, consider the set of vertices denoted as V_1, where each vertex $w \in V_1$ initially has a $L(i, w, O_{\alpha=i})$ value greater than 0, which subsequently decreases to less than or equal to 0 during the update process. These vertices are identified in Algorithm 2 lines 6-14 and stored in the set C. The time

complexity associated with this part is $O(|N_1(\mathsf{AFF}_{(\alpha,\beta)}, G^+)| \cdot |\mathsf{AFF}_{(\alpha,\beta)}|)$. Second, consider the set of vertices denoted as V_2, which satisfies the following conditions: for any $w \in V_2$, (1) w is not in the candidate set; (2) $u \preceq_{\alpha=i} w$ in $O_{\alpha=i}$; (3) there exists a sequence of edges $(u, w_1), (w_1, w_2), ..., (w_l, w)$ in G^+ from u to w where $w_i \in V_1$ for $1 \le i \le l$. The vertices in V_2 are identified in Algorithm 2 lines 15-21. Since V_2 belongs to the 1-hop neighbor set of $\mathsf{AFF}_{(\alpha,\beta)}$, the time complexity for this part can be derived as follows:

$$\sum_{w \in V_2} O(d(w)) \le \sum_{w \in V_2} O(|N_2(\mathsf{AFF}_{(\alpha,\beta)}, G^+)|)$$
$$= O(|N_1(\mathsf{AFF}_{(\alpha,\beta)}, G^+)| \cdot |N_2(\mathsf{AFF}_{(\alpha,\beta)}, G^+)|).$$

Therefore, the time complexity when considering a specific target c-pair (α, β) is bounded by $O(|N_1(\mathsf{AFF}_{(\alpha,\beta)}, G^+)| \cdot |N_2(\mathsf{AFF}_{(\alpha,\beta)}, G^+)|)$. Considering the processing of all c-pairs of u and v, the overall algorithm is bounded by $O(|N_1(\mathsf{AFF}, G^+)| \cdot |N_2(\mathsf{AFF}, G^+)|)$. Above all, the theorem is proved. □

A.8 Proof of Lemma 6.1

PROOF. if $A_{\alpha=i}(u, G) \le A_{\alpha=i}(v, G)$, then v is a neighbor of u in the $(i, A_{\alpha=i}(u, G))$-core before edge deletion. After the deletion of (u, v), v is no longer a neighbor of u in the $(i, A_{\alpha=i}(u, G))$-core, which results in $S(i, u, O_{\alpha=i})$ decreasing by 1. If $S(i, u, O_{\alpha=i}) \ge i$ after decreasing, u still has enough neighbors in the $(i, A_{\alpha=i}(u, G))$-core. Therefore, u cannot be added to the candidate set. If $A_{\alpha=i}(u, G) \ge A_{\alpha=i}(v, G)$, the proof follows similarly, so we omit the proof here. If neither u nor v is added to the candidate set, no update will occur. □

A.9 Proof of Lemma 6.2

PROOF. For condition (1), suppose vertex w is in the candidate set with $A_{\alpha=i}(w, G) < A_{\alpha=i}(u, G)$. This implies w loses neighbors from the $(i, A_{\alpha=i}(w, G))$-core after edge deletion. Observed that there is no vertex w' such that $w' \in N(u, G)$ and $A_{\alpha=i}(u, G^-) < A_{\alpha=i}(w', G) < A_{\alpha=i}(u, G)$. If $w \in N(u, G)$, then $A_{\alpha=i}(w, G) \le A_{\alpha=i}(u, G^-)$, keeping u inside the $(i, A_{\alpha=i}(w, G))$-core after edge deletion. If v is in the $(i, A_{\alpha=i}(w, G))$-core before edge deletion, it stays in after edge deletion. If $w \notin N(u, G)$, $(i, A_{\alpha=i}(w, G))$-core won't be affected after edge deletion because each neighbor $x \in N(u, G)$ with $A_{\alpha=i}(x, G) = A_{\alpha=i}(u, G)$ can drop by at most 1 (consider deleting u entirely). w cannot be in the candidate set. Suppose w is in the candidate set with $A_{\alpha=i}(w, G) > A_{\alpha=i}(u, G)$. This implies u is not part of $(i, A_{\alpha=i}(w, G))$-core initially. If v is in the $(i, A_{\alpha=i}(w, G))$-core before edge deletion, it stays in after edge deletion. Thus, w doesn't lose neighbor from $(i, A_{\alpha=i}(w, G))$-core. Consequently, vertex w cannot be included in the candidate set.

For condition (2), the updates begin with u (or v if $A_{\alpha=i}(u, G) = A_{\alpha=i}(v, G)$) and propagate locally. Therefore, if such a path exists, the updates can propagate to w locally; otherwise, w cannot be in the candidate set. □

A.10 Proof of Theorem 6.4

PROOF. As we discussed above, only vertices in CHANGED can be pushed into the queue, and each of them can be pushed exactly once. So, the bi-core maintenance process takes $O(|\text{CHANGED}| + |N_1(\text{CHANGED}, G)|)$ time. Since the BD-Order, the lack value, and the support value also should be maintained in the algorithm. Given CHANGED, these can be done in $O(|\text{CHANGED}| + |N_1(\text{CHANGED}, G)|)$ without increasing the time complexity of the deletion algorithm. So, above all, the time complexity of the proposed edge deletion algorithm is $O(|\text{CHANGED}| + |N_1(\text{CHANGED}, G)|)$, and the edge deletion algorithm is bounded. □
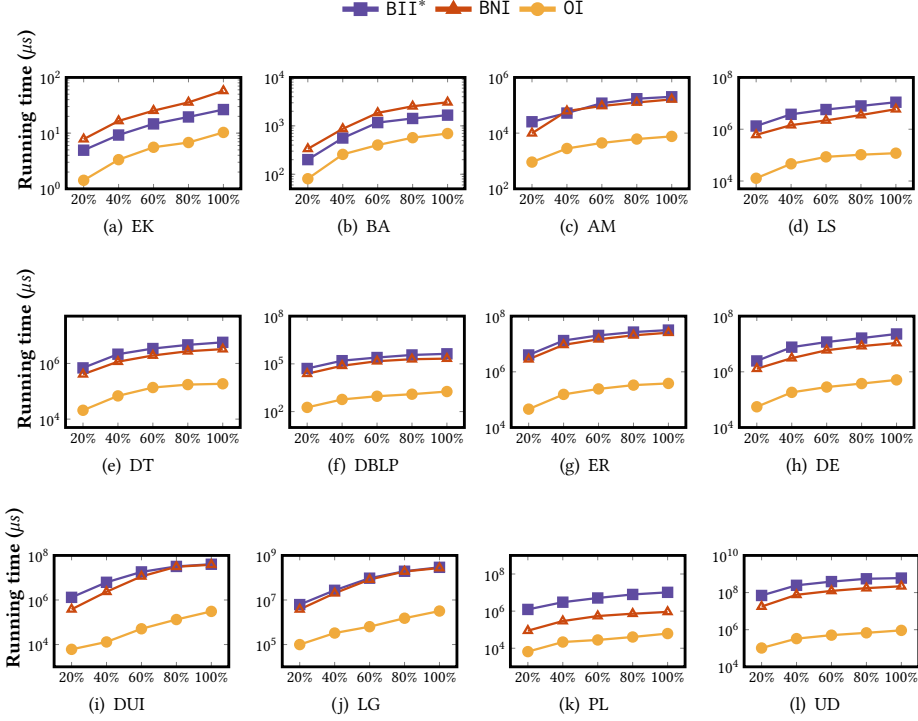
Fig. 17. Scalability test for handling edge insertions.

B Additional experimental results

• **Effect of the ratio of inserted edges $|\Delta E|/|E|$.** Figure 19 shows the ratio-based setting, where $r \in \{1\%\,|E|, 2\%\,|E|, 3\%\,|E|, 4\%\,|E|, 5\%\,|E|\}$. Note that the running time of Re-compute is stable as it needs to decompose the entire graph, so we omit it here. The results show that as the number of edges increases, the average insertion time of each edge does not increase significantly. The consistent behavior in both absolute-count and ratio-based settings demonstrates the stability and scalability of our method under varying insertion sizes.

• **Scalability test for edge insertion and deletion.** To evaluate scalability, we randomly selected 20%, 40%, 60%, 80%, and 100% of the edges from each graph to construct five induced subgraphs. For edge insertions and deletions, we conducted experiments on all datasets. As illustrated in Figs. 17 and 18, the execution time of all algorithms increases with the number of edges. However, our order-based algorithms consistently outperform BII* and BNI in edge insertions, as well as BIR* and BNR in edge deletions, across all settings. These results demonstrate that our maintenance algorithm achieves superior scalability in both insertion and deletion scenarios.
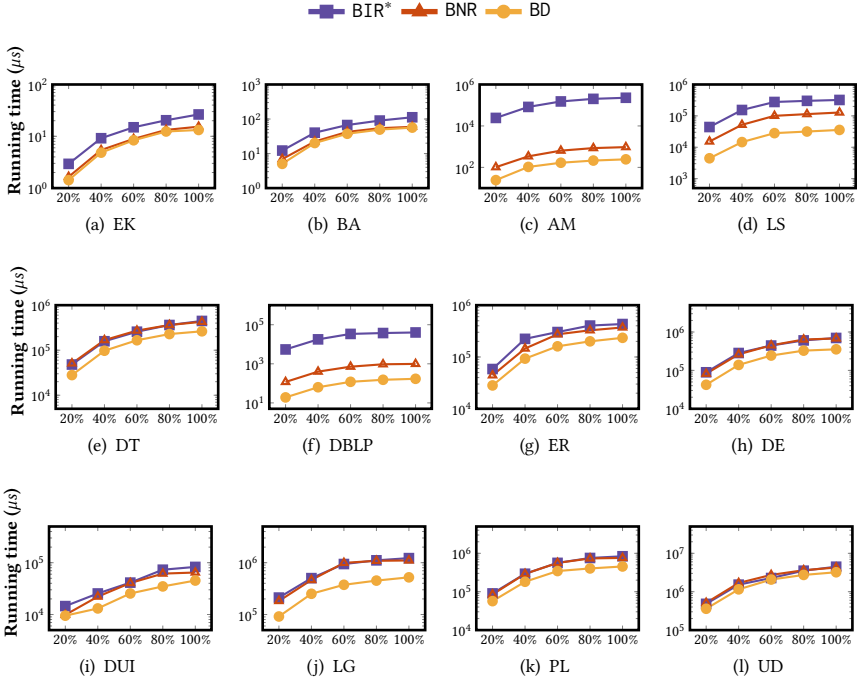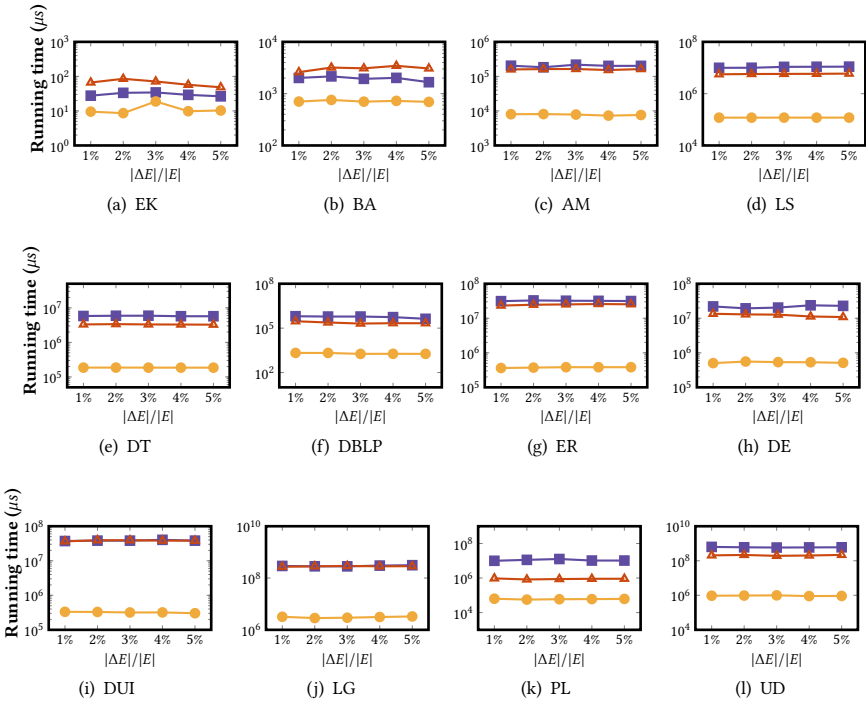
Fig. 18. Scalability test for handling edge deletions.

Fig. 19. Effect of the ratio of inserted edges $|\Delta E|/|E|$ (average time per edge).