An Order-based Approach for Efficient Core Maintenance in Large Bipartite Graphs

Qiaoyuan Yang
The Chinese University of Hong Kong, Shenzhen
qiaoyuanyang@link.cuhk.edu.cn

Wensheng Luo
The Chinese University of Hong Kong, Shenzhen
luowensheng@cuhk.edu.cn

Yixiang Fang
The Chinese University of Hong Kong, Shenzhen
fangyixiang@cuhk.edu.cn

Yuanyuan Zeng
The Chinese University of Hong Kong, Shenzhen
zengyuanyuan@cuhk.edu.cn

ABSTRACT

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 costly. Despite recent efforts to address bi-core maintenance in dynamic bipartite graphs, existing approaches lack theoretical analysis regarding the changes in bi-cores corresponding to graph changes, and also struggle to cope with the scale and frequency of these changes. To tackle these issues, we present efficient bi-core maintenance algorithms with theoretical guarantees in this paper. Specifically, we conduct boundedness analysis, which is a powerful tool for theoretically analyzing incremental algorithms over dynamic graphs, for 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 edge insertions, we propose a novel structure called the BD-Order, which transforms the solution into a relatively 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. Besides, for edge deletions, we develop a bounded algorithm based on 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.

PVLDB Reference Format:
Qiaoyuan Yang, Wensheng Luo, Yixiang Fang, and Yuanyuan Zeng. An Order-based Approach for Efficient Core Maintenance in Large Bipartite Graphs. PVLDB, 14(1): XXX-XXX, 2020.

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://anonymous.4open.science/r/Bicore-Maintenance-Order-E47C.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

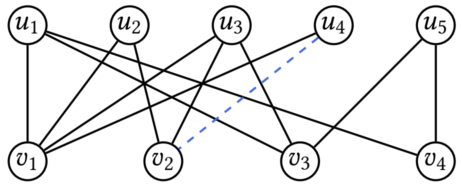

Figure 1: A bipartite graph G.

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 [11], collaboration networks [8], and gene co-expression networks [21]. For example, 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 [6, 24, 27], bitruss [38], and biclique [28]. 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 $\{u_1, u_2, u_3, u_5, v_1, v_2, v_3, v_4\}$ is the $(2, 2)$-core since the degrees of vertices on both sides of the subgraph are at least 2. Bi-core plays a crucial role in many applications such as graph visualization [1], e-commerce recommendations [16], and fraud detection in financial systems [3, 8, 40].

Efficiently computing all the bi-cores in a bipartite graph, called bi-core decomposition, has been extensively studied recently [10, 16, 23, 37]. While existing algorithms have demonstrated commendable computational efficiency, they are tailored for static graphs. In practical scenarios, graphs are frequently changed, implying that vertices and edges within the graph may be inserted or deleted. For instance, Amazon Logistics processed 4.79 billion U.S. delivery orders in 2022, equivalent to 13.13 million delivery orders per day or 546,941 per hour[1]. These orders result in a large dynamic bipartite graph comprising users and producers. As a result, the

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

bi-cores will be changed accordingly, which may further affect the downstream applications that benefit from bi-cores. For example, in recommendation networks, users' preferences for items constantly change, leading to varying preferences for different items over time. Therefore, it is crucial to investigate efficient algorithms for maintaining bi-cores in dynamic bipartite graphs, known as the problem of *bi-core maintenance*.

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 [16, 19, 30]. 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, 8, 40]. 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 problems in dynamic bipartite graphs. This includes identifying bicliques [25, 29, 43], computing the maximum k-biplex [26], and analyzing the hierarchical structure of bipartite graphs [39]. Efficient bi-core maintenance enhances the effectiveness of analysis and problem-solving in these bipartite networks.

A naive algorithm of bi-core maintenance is to recompute all bi-cores from scratch whenever an edge is inserted or deleted. This, however, is costly, requiring $O(m\sqrt{m})$ time [23], where m is the number of edges in the graph. In real-world scenarios, the numbers of changed vertices and edges are often much smaller than those of the entire graph, e.g., English Wikipedia had a 2.6% increase of articles in 2023[2]. Since these changes are small, the changes of bi-cores are also small. Based on the above observation, Liu et al. [24] introduced an algorithm that leverages local subgraphs near inserted or removed edges, significantly outperforming the above algorithm. However, it still involves checking numerous bi-cores, leading to huge redundant computations. Subsequently, Luo et al. [27] introduced the concept of bi-core number by exploiting the partially nested property of bi-cores, and developed algorithms to narrow down update scopes by analyzing the effect of edge insertion and deletion on vertex bi-core numbers, reducing computational redundancy and enhancing algorithm efficiency. Although these algorithms have made notable advancements, they face several limitations: They still involve redundant computations when traversing vertices potentially affected by graph changes via BFS (Breadth-First Search). Moreover, the lack of boundedness analysis on visited vertex numbers during this process implies that there is no theoretical guarantee regarding the relationship between the number of visited vertices and the number of edge insertions or deletions. Hence, there is much room to improve the efficiency of bi-core maintenance in both theories and algorithms.

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 [17]. *A maintenance algorithm is considered bounded if the cost of maintaining results is polynomial to graph updates; 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. [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. An unbounded incremental problem is considered still efficiently solvable if there is a relatively bounded incremental algorithm [17]. Therefore, to efficiently maintain bi-cores, our objective is to develop a relatively bounded algorithm for the case of edge insertions, and a bounded algorithm for the case of 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 peeled during bi-core decomposition. 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 (without considering the blue line), if $\alpha = 2$, the associated order could be $\{v_2, v_4, u_2, u_5, v_1, v_3, u_1, u_3\}$. This order signifies the peeling sequence of all vertices during the computation of all $(2, \cdot)$-cores.

Building upon the BD-Order, we introduce a novel concept of *lack value* for each vertex in the graph, which indicates the number of neighbors required for the vertex to join a new bi-core. Subsequently, we propose an efficient order-based algorithm for managing edge insertions. Our algorithm works by precisely identifying vertices that may be affected by the insertions using the BD-Order. Then, it decides whether to update the c-pairs of these vertices based on their lack values, where a c-pair for a vertex is an integer pair (α, β) that corresponds to the bi-core in which the vertex resides. This order-based approach reduces the number of visited vertices and minimizes neighbor information computation, thereby enhancing algorithm efficiency. We further analyze the theoretical complexity of our proposed order-based edge insertion algorithm and establish its relative boundedness.

For edge deletions, we introduce a novel concept called *support value* for each vertex, which is similar to the lack value, indicating the number of neighbors of a vertex within a certain bi-core. Utilizing the support value, we propose an efficient bounded algorithm for handling edge deletions. Unlike the algorithm for edge insertion, where the BD-Order accelerates the process, maintaining bi-cores with edge deletion being a bounded problem does not require direct usage of the BD-Order. Instead, we focus on preserving this structure. Furthermore, we analyze the complexity of this algorithm and provide a theoretical guarantee.

(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

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

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 [5].

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 ((α, β)-core [16]). *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, $\forall u \in U', d(u, G') \geq \alpha \wedge \forall v \in V', d(v, G') \geq \beta$.*

For any vertex in the (α, β)-core, we say that it has a *c-pair* (α, β). For a specific value of α, the maximum value of β such that the vertex u is included in the corresponding (α, β)-core of G is denoted as $B_u(\alpha, G)$. If no such (α, β)-core contains u, then $B_u(\alpha, G) = 0$. The case with $\alpha = 0$ is disregarded, as it holds no meaningful significance. Similarly, for a specified value of β, the corresponding value of α is denoted as $A_u(\beta, G)$. 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, $G_{\beta=i}$ is the $(0, i)$-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 ([24, 27]). *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 [24, 27]). *Given a bipartite graph G and all its bi-cores, update these bi-cores after the insertion or deletion of an edge (u, v).*

Since all bi-cores can be readily obtained by using the c-pairs of all the vertices, we maintain all the bi-cores by maintaining the c-pairs of all the vertices. Note that in line with previous works [24, 27], 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 insertion and edge deletions, respectively.

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 $V' \subseteq V$
$B_u(\alpha, G)$	The maximum value of β regarding α such that u is in the (α, β)-core in G
$A_u(\beta, G)$	The maximum value of α regarding β such that u is in the (α, β)-core in G
$G_{\alpha=i}, G_{\beta=i}$	The $(i, 0)$-core and $(0, i)$-core of G, respectively

3 ANALYSIS OF PROBLEM AND ALGORITHMS

In this section, we theoretically analyze the boundedness of bi-core maintenance problem and the limitations of existing algorithms.

3.1 Boundedness Analysis

We now introduce some notations and concepts of boundedness, and then present our theoretical analysis 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.

If ΔG (e.g., an inserted or deleted edge) is the input update to G, applying ΔG to G results in the updated query $Q(G \oplus \Delta G)$, where $G \oplus \Delta G$ means the updated G by applying ΔG to G. Let $D(G \oplus \Delta G)$ represent computing $Q(G \oplus \Delta G)$ from scratch using algorithm D, and $M(G, \Delta G)$ denote using algorithm M to incrementally compute $Q(G \oplus \Delta G)$. The output change is denoted as ΔR, leading to the equation $Q(G \oplus \Delta G) = Q(G) \oplus \Delta R$.

• **Concepts of boundedness.** The notion of *boundedness* [31] evaluates the effectiveness of an incremental algorithm using the metric CHANGED, defined as CHANGED $= \Delta G + \Delta R$, which leads to $|\text{CHANGED}| = |\Delta G| + |\Delta R|$. Clearly, this metric encompasses changes to both the input and output, reflecting the necessary work for the algorithm.

Definition 2 (Boundedness [17, 31]). *An incremental algorithm is bounded if its computational cost can be expressed as a polynomial function of* $|\text{CHANGED}|$ *and* $|Q|$. *Otherwise, it is unbounded.*

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

• **Our analysis.** Inspired by existing bi-core maintenance algorithms, we introduce the computation model for the incremental algorithm, specifically focusing on a class of algorithms known as *locally persistent* (LP) algorithms. LP algorithms were originally defined by Alpern et al. [4] as the underlying model of computation and have been studied in prior works on boundedness analysis [4, 17, 18, 31, 44].

An LP algorithm [4, 31] maintains a storage block for each edge, which includes pointers to adjacent edges and auxiliary status information but excludes pointers to non-adjacent edges. Additionally, no global auxiliary information is retained between successive calls to the algorithm; persistence is localized to the storage blocks of the edges. For the inserted or deleted edges, an LP algorithm starts with pointers to these newly inserted or deleted edges and traverses the graph using these pointers. The subsequent pointer to follow is deterministically chosen based on the status of the visited edges. Furthermore, the status of each edge is dynamically updated according to the status of the previously visited edges.

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 [5].

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 polynomial to* |CHANGED|*, making it bounded. The complete proof is in our full paper [5].*

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 [36], where both insertions and deletions are unbounded.

• **Concepts of relative boundedness.** In real-world graphs, |CHANGED| is often small, so some unbounded algorithms can be solved in polynomial time using measures comparable to |CHANGED|, making these algorithms feasible. To assess these incremental algorithms effectively, Fan et al. [17] introduced the concept of *relative boundedness*. While this concept addresses fully unbounded incremental problems, where both edge insertions and deletions are unbounded, it falls short for asymmetric incremental problems (i.e., problems that are unbounded for either edge insertions or deletions, and are bounded for the other) [44]. To address this limitation, Zhang et al. [44] introduced a revised concept of relative boundedness suitable for asymmetric incremental problems.

Given the asymmetric nature of the bi-core maintenance problem, we apply the revised concepts of relative boundedness [44]. Before delving into relative boundedness, we first introduce a concept called *affected part* processed by D, denoted by AFF.

Definition 3 (AFF [44]). *Given a graph G, a query Q, and the input updates ΔG to G,* AFF *signifies the cost difference of D between computing $Q(G)$ and $Q(G \oplus \Delta G)$.*

AFF captures the differences of executing the incremental algorithm. For unbounded problems, to evaluate the performance of the incremental algorithm, AFF provides a more accurate tool than CHANGED since it encompasses CHANGED, and accurately represents the necessary cost for incrementalizing D. If M is the incremental algorithm that incurs the minimum cost concerning AFF, then M is considered a relatively bounded incremental algorithm for D. This leads to the following definition of relative boundedness.

Definition 4 (Relative boundedness [44]). *An incremental graph algorithm M for Q is relatively bounded to D if its cost is polynomial in $|Q|$ and* |AFF|.

As stated in [17], relative boundedness is specific to a particular recomputation algorithm for a query, rather than all possible ones. Even if an incremental problem lacks boundedness, a relatively bounded algorithm M may still exist, optimized relative to the recomputation algorithm D with the same auxiliary structure.

3.2 Analysis of Existing Algorithms

By analyzing the two existing bi-core maintenance algorithms [24, 27], we observe that they generally follow the same framework from a high-level perspective. This framework consists of 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 updated c-pairs.
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 updated c-pair;
6      while $S \neq \emptyset$ do
7          $w \leftarrow S$.pop();
8          foreach $w' \in N(w, G)$ do
9              if $(u, v)$ is an insertion edge then
10                 $S$.push($w'$), $C$.insert($w'$);
11                 if $w' \notin (\alpha', \beta')$-core then $C$.erase($w'$) ;
12             else if $(\alpha, \beta)$ is no longer a c-pair of $w'$ then
13                 $S$.push($w'$), $C$.insert($w'$);
14     $(\alpha', \beta') \leftarrow$ the updated c-pair;
15     Update c-pairs of vertices in $C$ with $(\alpha', \beta')$;
```

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 updated c-pair, then checks if the neighbor vertex w' belongs in the bi-core of (α, β); if not, w' is removed from C (lines 4-11). For edge deletion, the algorithm adds vertices no longer in the bi-core of the current c-pair to C and then obtains the updated c-pair (lines 12-14). This process continues until S is empty, forming a BFS traversal. Finally, the c-pairs of the vertices in C are updated accordingly (line 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 $O(\sum_{\alpha=1}^{d(u,G)} |(\alpha, B_u(\alpha, G))\text{-core}|) + \sum_{\beta=1}^{d(v,G)} |(A_v(\beta, G), \beta)\text{-core}|$ time. When handling a deleted edge (u, v), the time complexity of Algorithm 1 is $O(|\mathsf{CHANGED}| + |N_1(\mathsf{CHANGED}, G)|)$.*

By Theorem 3.3, the existing algorithms for bi-core maintenance with edge deletions are bounded, while the algorithms for edge insertions are unbounded, and relatively unbounded due to the underutilization of auxiliary structures obtained during decomposition. The removal-friendly nature of bi-core maintenance is evident in the ease of incrementalization for edge deletions, as vertex's c-pairs can be naturally computed using peeling algorithms (e.g., bi-core decomposition algorithms [24]).

• **Limitations.** Existing algorithms focus on efficiently identifying the c-pairs to be analyzed. Liu et al. [24] 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. [27] 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.

Overall, existing algorithms primarily focus on improving line 1 of Algorithm 1. However, the BFS-manner process (lines 3-13) 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 [27] 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. Our objective in this work is to improve the above framework by proposing a relatively bounded algorithm for bi-core maintenance with edge insertions, with theoretical guarantees.

4 BI-CORE DECOMPOSITION ORDER

In this section, we introduce a novel concept called bi-core decomposition order (BD-Order), which serves as a cornerstone of our approach. We then formulate its AFF for theoretical analysis.

Bi-core Decomposition Order (BD-Order). The BD-Order represents the set of sequences in which vertices are removed during bi-core decomposition in a bipartite graph. Therefore, it can be obtained as a byproduct of the bi-core decomposition process without incurring any additional time cost. Since the bi-core is a two-dimensional model, the BD-Order in a bipartite graph is represented as a two-dimensional structure. For instance, for a positive integer i, all $(i, \cdot)$-cores can be computed by removing vertices from the entire graph in one pass during the bi-core decomposition algorithm, resulting in a sequence denoted as $O_{\alpha=i} = \{w_1, w_2, w_3, ..., w_{|G_{\alpha=i}|}\}$, where $|G_{\alpha=i}|$ is the number of vertices in $(i, 0)$-core of G. If a vertex w_k is removed before w_l in this decomposition process for a given i, it is denoted as $w_k \preceq_{\alpha=i} w_l$. When fixing the value of β, the same logic applies. Therefore, for all possible values of α and β, we derive a set of sequences forming the BD-Order for the entire graph.

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%

Note that for a bipartite graph, it's unnecessary to compute all the bi-cores corresponding to α and β from 0 to $\alpha_{\max}$ and $\beta_{\max}$, respectively. Liu et al. [24] introduced a bi-core decomposition algorithm, computing δ as the maximum value for which a (δ, δ)-core exists in G. Then, it iteratively computes all bi-cores in the graph by fixing α and β from 1 to δ. Following this decomposition approach, the BD-Order of a bipartite graph comprises δ sequences for both α and β, defined as follows.

Definition 5. *In a bipartite graph G, the BD-Order of G, denoted as O_G, is a list of sequences where each sequence represents the vertex removal order during bi-core decomposition for each fixed α and β. The combined BD-Order indicates the entire decomposition process of G. Specifically, O_G=$\{O_{\alpha=1}, O_{\alpha=2}, \cdots, O_{\alpha=\delta}, O_{\beta=1}, O_{\beta=2}, \cdots, O_{\beta=\delta}\}$, encapsulates the removal process for all specified α and β values.*

Table 3: The BD-Order for graph G.

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

In the subsequent discussion, each element of O_G (e.g., $O_{\alpha=i}$) is referred to as an order for clarity. Using BD-Order, we define lack value for each vertex as follows:

Definition 6 (Lack Value). *Given a bipartite graph G and an integer i, for a vertex w in the order $O_{\alpha=i}$, the lack value of w, denoted as $lack_{\alpha=i}(w)$, is defined as*

$$lack_{\alpha=i}(w) = B_w(i, G) + 1 - \left|\{w' \in N(w, G) \mid w \preceq_{\alpha=i} w'\}\right|.$$

The lack value indicates a vertex's "position" in an order, representing the number of neighbors removed after it during the bi-core decomposition process.

Example 1. *In Figure 1, for graph G (excluding edge (u_4, v_2)), $\delta = 2$. After bi-core decomposition, the BD-Order is presented in Table 3. Focusing on $\alpha = 2$, where $O_{\alpha=2} = \{v_2, v_4, u_2, u_5, v_1, v_3, u_1, u_3\}$, the* *corresponding lack values for these vertices are 1, 1, 1, 1, 1, 1, 2, and 2, respectively, as per Definition 6. For vertices in other orders of O_G, lack values can be derived similarly.*

Then we observe a property regarding order and lack value.

Property 2. *For a bipartite graph $G = (U, V, E)$,*

- *given a fixed $\alpha = i$, a sequence O comprises the vertices on $G_{\alpha=i}$. The sequence O qualifies as a BD-Order element if and only if: (1) for any two vertices w and w', if $B_w(i,G) < B_{w'}(i,G)$, then $w \preceq_{\alpha=i} w'$ and (2) $\forall w \in U, 0 < lack_{\alpha=i}(w) \leq i$; $\forall w \in V, 0 < lack_{\alpha=i}(w) \leq B_w(i,G)+1$.*
- *given a fixed $\beta = i$, a sequence O comprises the vertices on $G_{\beta=i}$. The sequence O qualifies as a BD-Order element if and only if: (1) for any two vertices w and w', if $A_w(i,G) < A_{w'}(i,G)$, then $w \preceq_{\beta=i} w'$ and (2) $\forall w \in U, 0 < lack_{\beta=i}(w) \leq A_w(i,G)+1$; $\forall w \in V, 0 < lack_{\beta=i}(w) \leq i$.*

Formulating 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. Let ΔG be a set of edges to be inserted into G. Consider BD-Orders O_G and $O'_{G\oplus\Delta G}$ for graphs G and $G \oplus \Delta G$, respectively. We measure the difference between these two BD-Orders as: $\texttt{diff}(O_G, O'_{G\oplus\Delta G}) = \{\cup_{i=1}^{\delta}[\texttt{diff}(O_{\alpha=i}, O'_{\alpha=i}) \cup \texttt{diff}(O_{\beta=i}, O'_{\beta=i})]\}$, where $\texttt{diff}\ (O_{\alpha=i}, O'_{\alpha=i}) = \{w \in G_{\alpha=i} | \exists w' \in G_{\alpha=i}, w \preceq_{\alpha=i} w' \wedge w' \preceq'_{\alpha=i} w\}$, in other words, it includes vertices whose positions in $O'_{\alpha=i}$ have shifted backward relative to their positions in the original order $O_{\alpha=i}$.

$\texttt{diff}(O_{\beta=i}, O'_{\beta=i})$ can be represented in the same way. Note that $O'_{G\oplus\Delta G}$ is not unique. Therefore, for BD-Order, AFF is the one with the minimum size among all possible $\texttt{diff}(O_G, O'_{G\oplus\Delta G})$.

AFF encompasses two key aspects: (1) it includes the input and output changes denoted by CHANGED after inserting edges, and (2) it comprises vertices $w \in G$ whose removal order in the peeling process requires adjustment after applying ΔG to G, ensuring that vertices originally positioned after w in $O_{\alpha=i}$ or $O_{\beta=i}$ are removed before w to maintain consistency. Notably, AFF serves as a lower bound, representing the necessary cost for updating the bi-core decomposition.

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 $\{v_2, v_4, u_2, u_5, v_1, v_3, u_1, u_3\}$. After insertion, the new order $O'_{\alpha=2}$ becomes $\{v_4, u_5, v_3, u_1, v_1, v_2, u_4, u_2, u_3\}$. 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 the order-based bi-core maintenance algorithm tailored to handle edge insertion. Upon inserting an edge (u, v), the updated graph is denoted as G^+. Without loss of generality, we assume $u \in U$ and $v \in V$. The algorithm comprises two key steps: first, identifying the candidate set consisting of vertices needing updates to their c-pairs following the edge insertion; second, updating vertices' c-pairs while preserving the BD-Order for streamlined subsequent computations.

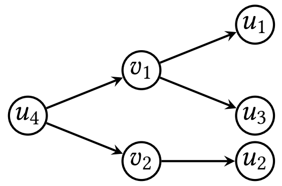

Figure 2: Paths where the vertices satisfy partial ordering relations in $O_{\alpha=2}$.

5.1 Analytical Scope of the Candidate Set

In Section 2, we show that the BFS process for identifying the candidate set consumes a significant portion of time in the existing edge insertion algorithms, as depicted in Algorithm 1 lines 3-13. Therefore, we introduce a novel algorithm based on BD-Order, aimed at efficiently identifying the candidate set. To achieve this, we first present the following lemma.

Lemma 5.1. *Given a bipartite graph G and an integer i, when inserting an edge (u,v) where $u \preceq_{\alpha=i} v$ according to the original order $O_{\alpha=i}$, whether vertex w is included in the candidate set depends on the following conditions:*

1. *If $B_w(i,G) < B_u(i,G)$, then w cannot be in the candidate set.*
2. *If $B_w(i,G) > B_v(i,G)$, then w cannot be in the candidate set.*
3. *If $B_u(i,G) \leq B_w(i,G) \leq B_v(i,G)$ and $w \preceq_{\alpha=i} u$, then w cannot be in the candidate set.*
4. *If $B_u(i,G) \leq B_w(i,G) \leq B_v(i,G)$ and $u \preceq_{\alpha=i} w$, w may be in the candidate set, provided there is a path $w_0, w_1, \cdots, w_t$ such that $w_0 = u$, $w_t = w$, $(w_i, w_{i+1}) \in E$, and $w_i \preceq_{\alpha=i} w_{i+1}$ for $0 \leq i < t$. Otherwise, w cannot be in the candidate set.*

Note that the fulfillment of condition (4) does not assure the inclusion of a vertex w in the candidate set. Similar conclusions hold for $\beta = i$ as long as we swap α and β, A and B respectively.

Example 3. *For the graph G depicted in Figure 1, considering the inserted edge (u_4, v_2) and order $O_{\alpha=2}$, it holds that $u_4 \preceq_{\alpha=2} v_2$ with $B_{u_4}(2,G) = 0$ and $B_{v_2}(2,G) = 2$. For conditions (1) or (2), no vertex w in G satisfies $B_w(2,G) < 0$ or $B_w(2,G) > 2$. For condition (3), with the original order $O_{\alpha=2} = \{v_2, v_4, u_2, u_5, v_1, v_3, u_1, u_3\}$ from Table 3, inserting edge (u_4, v_2) initially places u_4 at the beginning of the order since it has only one neighbor beforehand. Thus, no vertex meets the condition. For condition (4), Figure 2 depicts a collection of possible paths, where $w \rightarrow w'$ indicates that $(w, w') \in E$ and $w \preceq_{\alpha=2} w'$. Hence, apart from u_4 and v_2, v_1, u_1, u_3, and u_2 could also be potential candidates. However, u_5, v_3, and v_4 lack such a connecting path to u_4, excluding them from the candidate set.*

Note that upon the insertion of (u, v), $lack_{\alpha=i}(u)$ decreases by 1 to reflect the insertion, while $lack_{\alpha=i}(v)$ remains unchanged initially. If u still lacks sufficient neighbors appearing after u in the order $O_{\alpha=i}$ to support a change after edge insertion, meaning $lack_{\alpha=i}(u) > 0$ after the initial decrease, then no vertices in the original $(i,\cdot)$-cores can be included in the candidate set. Thus, we have the following lemma.

Lemma 5.2. *Given a bipartite graph G and an integer i, consider the insertion of an edge (u,v) into G, such that $u \preceq_{\alpha=i} v$. If the original $lack_{\alpha=i}(u)$ is greater than 1, then no vertex in the original $(i,\cdot)$-cores requires updating.*

Example 4. *In the graph shown in Figure 1, let (u_4, v_2) be the edge to be inserted. Considering $\alpha = 2$, we have $u_4 \preceq_{\alpha=2} v_2$. With u_4 confirmed in $(2,2)$-core, we proceed to verify whether it belongs to $(2,3)$-core, at which point $lack_{\alpha=2}(u_4) = 1$, indicating that the update can occur. After the update, u_4 is added to the $(2,3)$-core in G^+, and $lack_{\alpha=2}(u)$ is updated to 2. Therefore, even if u_4 establishes a new link with any other vertex, it cannot be added to the $(2,4)$-core.*

Lemma 5.2 indicates the necessity of updating the $(i,\cdot)$-cores. Upon satisfying this necessity, we explore the scope of potential updates for vertices in the subsequent lemma, focusing on vertices specified in condition (4) of Lemma 5.1.

Lemma 5.3. *After inserting an edge (u,v) with $u \preceq_{\alpha=i} v$ into a bipartite graph G for a given $\alpha = i$.*

- *For vertex u, if $B_u(i,G)$ is to be updated, the range for the updated value $B_u(i,G^+)$ is $(B_u(i,G), B_v(i,G)+1]$.*
- *For vertex v, if $B_v(i,G)$ is to be updated, the updated value $B_v(i,G^+)$ is $B_v(i,G)+1$;*
- *For any vertex w except u and v, if $B_w(i,G)$ is to be updated, the updated value $B_w(i,G^+)$ is $B_w(i,G)+1$.*

Example 5. *Given the graph in Figure 1, let (u_4, v_2) be the edge to be inserted. Considering $\alpha = 2$, we have $u_4 \preceq_{\alpha=2} v_2$. According to Lemma 5.3, the scopes of $B_{u_4}(2,G^+)$ and $B_{v_2}(2,G^+)$ are $(0,3]$ and $[2,3]$, respectively. For any other vertex, such as u_2, we have $B_{u_2}(2,G) = 2$. Therefore, if u_2 is to be updated, the updated value can only be $B_{u_2}(2,G)+1 = 3$.*

Lemma 5.3 demonstrates that after inserting the edge (u,v) with $u \preceq_{\alpha=i} v$ for $\alpha = i$, only the value $B_u(i,G^+)$ of vertex u needs to be determined if an update is necessary. For other vertices, the update will only result in their values increasing by one.

5.2 Order-based Candidate Set Identification

Leveraging the lemmas in Section 5.1 allows for precise determination of the scopes of the vertices to be updated and their updated values after inserting edge (u,v) in G. In this section, we conduct a thorough analysis to identify the candidate set, which comprises the vertices requiring updates to their c-pairs.

Initially, let (i,j) be the target c-pair for an update currently being discussed. Given the order $O_{\alpha=i}$ and $u \preceq_{\alpha=i} v$, for each vertex with $B_w(i,G) = j-1$, we have $lack_{\alpha=i}(w) = j - |\{w' \in N(w,G) \wedge w \preceq_{\alpha=i} w'\}|$. Then, starting from u, we visit vertices iteratively by traversing the current vertex's neighbors in $\{w | B_w(i,G) = j-1 \wedge u \preceq_{\alpha=i} w\}$. During the visiting process, we first compute the updated lack values for all visited vertices according to the original order, subsequently identifying the candidate set based on these values. Specifically, for a vertex w, if $lack_{\alpha=i}(w) > 0$, then it will not be a candidate vertex; if $lack_{\alpha=i}(w) \leq 0$, we consider the vertex set $P = \{w' \in N(w,G^+) | lack_{\alpha=i}(w') > 0 \wedge w \preceq_{\alpha=i} w' \wedge B_{w'}(i,G) = j-1\}$; when $lack_{\alpha=i}(w) + |P| \leq 0$, w should be in the candidate set; otherwise, w cannot be added to the candidate set.

Given a graph G, the insertion edge (u,v), and a target c-pair (i,j), Algorithm 2 outlines the procedure for identifying all vertices newly added to the (i,j)-core of G^+. In particular, a queue S is utilized to store the vertices to be visited, a set C represents the candidate set, and a set T stores the vertices eligible for potential inclusion in the (i,j)-core of G^+, pending further verification (line 1). If $i < j$, the algorithm utilizes order $O_{\alpha=i}$ subsequently (line 2). Initially, the algorithm starts with the vertex u, and $lack_{\alpha=i}(u)$ is decremented by 1 due to the edge insertion and the assumption that $u \preceq_{\alpha=i} v$ (line 3). Per Lemma 5.2, if $lack_{\alpha=i}(u)$ stays above zero, no update occurs (line 4). Otherwise, u enters both candidate set C and queue S (line 5). The algorithm then visits vertices in S sequentially (line 6), decrementing $lack_{\alpha=i}(w')$ by one for each neighbor w' of the current vertex w, where $B_{w'}(i,G) = j-1$ and $w \preceq_{\alpha=i} w'$ (lines 7-11). If $lack_{\alpha=i}(w') = 0$ after the decrement, w' is removed from T (if included), and then added to both S and C (lines 12-14). Otherwise, if $lack_{\alpha=i}(w') > 0$, w' is added to T (line 15).

Note that after the first while loop in Algorithm 2, some vertices in C may fail to meet the criteria for updates. To identify and remove these vertices from the candidate set, the algorithm first converts the set T into a queue S (line 16), and sequentially checks each vertex w in S to verify if their neighbors qualify for continued inclusion in the candidate set (lines 17-23). For each neighbor w' of w in C that precedes w in the order, its $lack_{\alpha=i}(w')$ increases by one (lines 19-21). After that, w' exits the candidate set and joins S if $lack_{\alpha=i}(w') > 0$ (lines 22-23). This process is iterated until S empties. If $i \geq j$, the operations are similar to lines 3-23 by swapping i with j, α with β, A with B respectively (lines 24-25).

Algorithm 2 relies on `OPrec`$(O_{\alpha=i}, w, w')$ to check if w precedes w' in $O_{\alpha=i}$. This, along with removing w and inserting it next to w' in $O_{\alpha=i}$, tackles the well-known *order maintenance problem* [7, 15], solvable in $O(1)$ time with linear space.

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

```
input  : $G$, $(u,v)$, target c-pair $(i,j)$
output : $C$ (all vertices newly included in $(i,j)$-core in $G^+$)
1  $S, C, T \leftarrow \emptyset$;
2  if $i < j$ then
3      $lack_{\alpha=i}(u) \leftarrow lack_{\alpha=i}(u) - 1$;        ▷ Assuming $u \preceq_{\alpha=i} v$
4      if $lack_{\alpha=i}(u) > 0$ then return;
5      $C$.insert($u$); $S$.push($u$);
6      while $S \neq \emptyset$ do
7          $w \leftarrow S$.pop();
8          for $w' \in N(w, G^+)$ do
9              if OPrec($O_{\alpha=i}, w', w$) then continue;
10             if $\mathcal{B}_{w'}(j,G) = j-1$ then
11                 $lack_{\alpha=i}(w') \leftarrow lack_{\alpha=i}(w') - 1$;
12                 if $lack_{\alpha=i}(w') = 0$ then
13                     if $w' \in T$ then $T$.erase($w'$);
14                     $S$.push($w'$); $C$.insert($w'$);
15                 else if $lack_{\alpha=i}(w') > 0$ then $T$.insert($w'$) ;
16     $S \leftarrow T$ ;
17     while $S \neq \emptyset$ do
18         $w \leftarrow S$.pop();
19         for $w' \in N(w, G^+)$ do
20             if $w' \in C \wedge$ OPrec($O_{\alpha=i}, w', w$) then
21                 $lack_{\alpha=i}(w') \leftarrow lack_{\alpha=i}(w') + 1$;
22                 if $lack_{\alpha=i}(w') > 0$ then
23                     $C$.erase($w'$); $S$.push($w'$);
24 else
25     lines 3-23 by swapping $i$ with $j$, $\alpha$ with $\beta$, $A$ with $B$.
```

Example 6. *Given the graph G in Figure 1, let (u_4, v_2) be the edge to be inserted. Consider the target c-pair $(2,3)$, $lack_{\alpha=2}(u_4) - 1 = 0$ with order $O_{\alpha=2}$ and $u_4 \preceq_{\alpha=2} v_2$, we start by adding u_4 to the candidate set C. Next, we compute $lack_{\alpha=2}(v_1) - 1 = lack_{\alpha=2}(v_2) - 1 = 0$ because v_1 and v_2 satisfies that (1) $\{v_1, v_2\} \in N(u_4, G^+)$; (2) $B_{v_1}(2, G) = B_{v_2}(2, G) = 2$; (3) $u_4 \preceq_{\alpha=2} v_1$, $u_4 \preceq_{\alpha=2} v_2$, leading to the addition of v_1 and v_2 into S and C. Then, we continue to process v_1's neighbors u_1 and u_3 due to the conditions in lines 9-10 of Algorithm 2. As $lack_{\alpha=2}(u_1) = lack_{\alpha=2}(u_3) = 1 > 0$, we add u_1 and u_3 to set T. When the queue S is empty, $C = \{u_4, v_1, v_2, u_2, u_3\}$, $T = \{u_1\}$, with lack values of vertices in C updated to 0,-1,0,0,0, respectively. Note that $lack_{\alpha=2}(u_3)$ is decreased to 0 when processing v_2, so it is removed from T and added into C. Subsequently, u_1 in T is processed. Among the neighbors of u_1, only v_1 meet the conditions in line 20 of Algorithm 2. This increase $lack_{\alpha=2}(v_1)$ to 0, retaining v_1 in the candidate set. Finally, algorithm terminates with $C = \{u_4, v_1, v_2, u_2, u_3\}$.*

5.3 BD-Order Maintenance

Suppose the target c-pair is (i, j) with $i < j$. After inserting (u, v) into G, the c-pair of vertices in the candidate set, except for u, changes from $(i, j-1)$ to (i, j). Hence, we need to update the corresponding order. Let the original order be $O_{\alpha=i}$, and given $u \preceq_{\alpha=i} v$, we use $O_{\alpha=i}(j)$ to denote the sequence of vertices in $O_{\alpha=i}$ whose $B_{(\cdot)}(i, G) = j$. For a vertex w in $O_{\alpha=i}(j)$, if $w \in U$, then $lack_{\alpha=i}(w) \leq i$; if $w \in V$, then $lack_{\alpha=i}(w) \leq j + 1$. We have a sequence of $O_{\alpha=i}(1), O_{\alpha=i}(2), \ldots$, and denote $O_{\alpha=i}(i) \preceq_{\alpha=i} O_{\alpha=i}(j)$ if $i < j$.

Since the c-pairs of the vertices in the candidate set except u changes from $(i, j-1)$ to (i, j), we need to generate two new sequences $O'_{\alpha=i}(j-1)$ and $O'_{\alpha=i}(j)$, composed respectively of vertices with $B_{(\cdot)}(i, G^+) = j-1$ and $B_{(\cdot)}(i, G^+) = j$. Then we can obtain the new order $O'_{\alpha=i}$ for G^+ by replacing the sequences $O_{\alpha=i}(j-1)$ and $O_{\alpha=i}(j)$ in the original order $O_{\alpha=i}$ with $O'_{\alpha=i}(j-1)$ and $O'_{\alpha=i}(j)$, rather than updating the entire order. Then, we discuss how to obtain $O'_{\alpha=i}(j-1)$ and $O'_{\alpha=i}(j)$.

Figure 3 illustrates the main idea of order maintenance with edge insertions, where each partition labeled with a number represents a sequence of vertices in the order. In the figure, vertices preceding u in $O_{\alpha=i}(j-1)$ (partition 1) remain unchanged in the updated order $O'_{\alpha=i}(j-1)$. For vertices after u (including u) in $O_{\alpha=i}(j-1)$, we divide them into three parts based on Algorithm 2, each depicted in a different color in the figure. The first part (partitions 6 and 11) consists of vertices in T after the first while-loop of Algorithm 2, maintaining their original relative positions. The second part (partitions 4 and 9) comprises vertices processed in lines 22-23 of Algorithm 2, retaining the resulting order from the algorithm and positioned after the relevant vertices in the first part in the new order $O'_{\alpha=i}(j-1)$. The third part (partitions 3, 5, 7, 10, and 12) includes the vertices not visited by Algorithm 2. For any vertex w in this part, if there exists a vertex w' belonging to the first or second parts such that $w \preceq_{\alpha=i} w'$, and w' is added to $O'_{\alpha=i}(j-1)$, then w is inserted before w' with its original relative position. For example, in Figure 3, partitions 3 and 5 belong to the third part, when partition 6 is added to $O'_{\alpha=i}(j-1)$, partitions 3 and 5 precede it with the original order.

For $O'_{\alpha=i}(j)$, we insert the vertices in the candidate set that were not originally in $O_{\alpha=i}(j)$ at the beginning of it while preserving their original order. For vertex u, we remove it from $O_{\alpha=i}(B_u(i, G))$

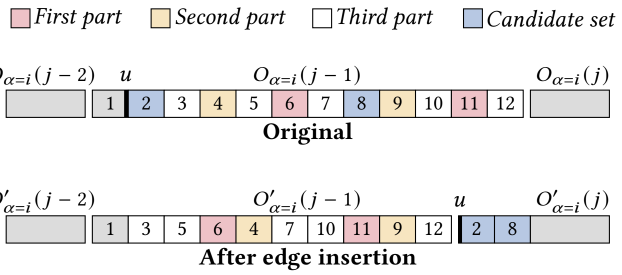

Figure 3: The idea of order maintenance with edge insertion.

and add it to $O'_{\alpha=i}(j)$ while preserving its relative position in $O_{\alpha=i}$ with other candidate vertices. If $i \geq j$, the operations are similar by swapping α with β, A with B respectively.

Example 7. *Reconsider Example 6, we maintain the order $O_{\alpha=2}$ as follows. Initially, $O_{\alpha=2}(2) = \{v_2, v_4, u_2, u_5, v_1, v_3, u_1, u_3\}$. With the insertion of edge (u_4, v_2), increasing u_4's degree from 1 to 2, we position u_4 at the beginning of $O_{\alpha=2}(2)$. Subsequently, in $O_{\alpha=2}$, following u_4, u_1 falls into the first part as a vertex in T after the first while-loop of Algorithm 2, while the second part remains empty due to the absence of vertex removals from the candidate set in the subsequent while-loop. The vertices in the third part are sequenced as $\{v_4, u_5, v_3\}$ under partial relation in $O_{\alpha=2}$. As all of them precede u_1 within $O_{\alpha=2}$, the revised order $O'_{\alpha=2}(2)$ is $\{v_4, u_5, v_3, u_1\}$. After inserting the vertices in the candidate set, $O'_{\alpha=2}(3) = \{u_4, v_2, u_2, v_1, u_3\}$. In general, $O'_{\alpha=2} = \{v_4, u_5, v_3, u_1, u_4, v_2, u_2, v_1, u_3\}$.*

5.4 Overall Edge Insertion Algorithm

Based on the proposed methods for identifying the candidate set and maintaining BD-Order, we can present the overall algorithm to handle edge insertions efficiently. The algorithm utilizes the framework in Algorithm 1, which provides the updated c-pair set P in line 1. Then, we employ Algorithm 2 to compute the candidate set instead of using Algorithm 2 lines 3-13. Since the predominant component contributing to the time complexity of Algorithm 1 is lines 3-13, our algorithm is capable of significantly improving performance outcomes. Then we conduct a comprehensive analysis of the algorithm, encompassing 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 5.4. *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 $lack_{\alpha=i}(w)$ is not less than its value before the update.*

Then we have Theorem 5.5 for the overall algorithm.

Theorem 5.5. *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. Given a graph G and an insertion edge (u, v), the time complexity of the overall algorithm can be described by the following theorem.

Theorem 5.6. *Given one inserted edge (u,v) in graph G, the time complexity of the proposed edge insertion algorithm is $O(|N_1(\mathsf{AFF}, G^+)| \cdot |N_2(\mathsf{AFF}, G^+)|)$, where $\mathsf{AFF} = \mathtt{diff}(O_G, O'_{G\oplus\Delta G})$.*

Note that from the complexity analysis, the proposed edge insertion algorithm visits only a subset of the vertices visited by the existing edge insertion algorithms, which explains its practical superiority.

Boundedness. Then, we analyze the boundedness of our proposed edge insertion algorithm.

Theorem 5.7. *The order-based bi-core maintenance algorithm with edge insertion is relatively bounded to the bi-core decomposition algorithm [24], since its cost can be expressed as a polynomial function of* AFF.

6 BOUNDED EDGE DELETION

In this section, we present an efficient bi-core maintenance algorithm for handling edge deletions. Upon deleting an edge (u,v), the updated graph is denoted as G^-. Without loss of generality, we assume $u \in U$ and $v \in V$. Like edge insertion, this process involves two main steps: first, identifying the candidate set of vertices needing c-pair updates due to edge deletion; then, we maintain the BD-Order to aid future computations. However, unlike the insertion algorithm, since bi-core maintenance with edge deletion is a bounded problem, there's no need to directly use the BD-Order in the first step; we simply need to preserve this structure.

6.1 Candidate Set Identification

To efficiently identify the candidate set when edges are deleted, we define the support value for each vertex as follows, similar to the lack value in edge insertion:

Definition 7 (Support Value). *Given a bipartite graph G and an order $O_{\alpha=i}$ where i is an integer, the support value of a vertex w in the order $O_{\alpha=i}$, denoted as $sup_{\alpha=i}(w)$, is defined as follows:*

$$sup_{\alpha=i}(w) = \left|\{w' \in N(w,G) \mid B_w(i,G) \le B_{w'}(i,G)\}\right|.$$

For a vertex w, its support value represents the number of its neighbors w' in G for which $B_w(\alpha,G) \le B_{w'}(\alpha,G)$. This value can be computed efficiently as a byproduct within the decomposition algorithm, without adding to its time complexity. Based on support value, we have the following lemma.

Lemma 6.1. *Given a graph G, a deleted edge (u,v), and an order $O_{\alpha=i}$,*

- *If $B_u(\alpha,G) < B_v(\alpha,G)$, then for any vertex w in the original $(\alpha,\cdot)$-cores, $B_w(\alpha,G^-)$ remains unchanged if $sup_{\alpha=i}(u) \ge \alpha$ after decreasing it by 1; similarly, if $B_u(\alpha,G) > B_v(\alpha,G)$, $B_w(\alpha,G^-)$ remains unchanged if $sup_{\alpha=i}(v) \ge B_v(\alpha,G)$ after decreasing it by 1.*
- *If $B_u(\alpha,G) = B_v(\alpha,G)$, then for any vertex w in the original $(\alpha,\cdot)$-cores, $B_w(\alpha,G^-)$ remains unchanged if $sup_{\alpha=i}(u) \ge \alpha$ and $sup_{\alpha=i}(v) \ge B_v(\alpha,G)$ after decreasing these two values by one, respectively.*

Example 8. *In the graph G shown in Figure 1 (contains the blue line here), let (u_3,v_3) be the edge to be deleted. We observe $B_{u_3}(2,G) = 3 > B_{v_3}(2,G) = 2$. Initially, $sup_{\alpha=2}(v_3) = 3$. Thus, upon decrementing, $sup_{\alpha=2}(v_3) = 2 = B_{v_3}(2,G)$, indicating that any vertex w in the original $(2,\cdot)$-cores has $B_w(2,G)$=$B_w(2,G^-)$.*

Algorithm 3: $\mathtt{Bounded\text{-}Del}(G,(u,v),(i,j))$

```
input  : $G$, $(u,v)$, and original c-pair $(i,j)$
output : C (vertices no longer in the $(i,j)$-core in $G^-$)
1  $S, C \leftarrow \emptyset$;
2  if $i < j$ then
3      if $B_u(i,G) \le B_v(i,G)$ then
4          $sup_{\alpha=i}(u) \leftarrow sup_{\alpha=i}(u) - 1$;
5          if $sup_{\alpha=i}(u) = i - 1$ then
6              $S$.push($u$); $C$.insert($u$);
7      if $B_u(i,G) \ge B_v(i,G)$ then
8          $sup_{\alpha=i}(v) \leftarrow sup_{\alpha=i}(v) - 1$;
9          if $sup_{\alpha=i}(v) = B_v(i,G) - 1$ then
10             $S$.push($v$); $C$.insert($v$);
11     while $S \neq \emptyset$ do
12         $w \leftarrow S$.pop();
13         for $w' \in N(w,G^-)$ do
14             if $B_{w'}(i,G) = j$ then
15                 $sup_{\alpha=i}(w') \leftarrow sup_{\alpha=i}(w') - 1$ ;
16                 if $sup_{\alpha=i}(w') = i - 1 \land w' \in U$ then
17                     $S$.push($w'$); $C$.push($w'$) ;
18                 if $sup_{\alpha=i}(w') = j - 1 \land w' \in V$ then
19                     $S$.push($w'$); $C$.push($w'$) ;
20 else
21     Lines 3-19 by swapping $i$ with $j$, $\alpha$ with $\beta$, $A$ with $B$.
```

Lemma 6.1 shows that updates may not always occur, reducing the scope of updates. Based on this, we propose Algorithm 3 to identify the candidate set after edge deletions for the original c-pair (i,j). Given a graph G, a deleted edge (u,v), and an original c-pair (i,j), Algorithm 3 initializes a queue S and a set C to track the vertices to be processed and the candidate vertices to be removed from the original (i,j)-core, respectively (line 1). If $i < j$, the algorithm begins with order $O_{\alpha=i}$. Firstly, it checks if $sup_{\alpha=i}(u) = i-1$ after the potential decrease, and if true, adds u to both S and C (lines 3-6). Then, it checks if $sup_{\alpha=i}(v) = B_v(i,G) - 1$ after the potential decrease, and if true, v is added to both S and C (lines 7-10). Subsequently, the algorithm iterates through the vertices in S, with w being the currently visited vertex (lines 11-12). For each neighbor w' of w where $B_{w'}(i,G) = j$, the algorithm decreases $sup_{\alpha=i}(w')$ by one (lines 13-15). If $w' \in U$ and $sup_{\alpha=i}(w')$ decreases to $i-1$, or if $w' \in V$ and $sup_{\alpha=i}(w')$ decreases to $j-1$, w' should be removed from the original (i,j)-core due to the deletion of w. In such cases, w' is added to both S and C (lines 16-19). The algorithm visits each vertex in S sequentially in the manner described above, until S is empty. If $i \ge j$, the operations are similar to lines 3-19 by swapping i with j, α with β, and A with B.

Example 9. *Consider graph G in Figure 1, with (u_4,v_2) as the edge to be deleted. Given the original c-pair $(2,3)$, we find $B_{u_4}(2,G) = B_{v_2}(2,G) = 3$. Setting $\alpha = 2$, we observe $sup_{\alpha=2}(u_4) = 1$ and $sup_{\alpha=2}(v_2) = 2$ after decreasing, leading to the addition of u_4 and v_2 into S and C. Subsequently, for neighbor v_1 of u_4, its support value decreases to 2, warranting its addition 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

Given the deleted edge (u, v), one original c-pair (i, j) and an order $O_{\alpha=i}$. Suppose $u \preceq_{\alpha=i} v$, for each w which belongs to the candidate set and $w \neq u$, we remove w from $O_{\alpha=i}(j)$, and insert it to the end of $O'_{\alpha=i}(j-1)$. This differs from the edge insertion case, where we insert vertices to the beginning of $O'_{\alpha=i}(j+1)$. u is a special case since the updated $B_u(i, G^-)$ may not be j–1. After removing u from $O_{\alpha=i}(j)$, it should be inserted to the end of $O'_{\alpha=i}(B_u(i, G^-))$.

Example 10. *Reconsider Example 9, we maintain the order $O_{\alpha=2}$ as follows. Initially, we have $O_{\alpha=2}(2) = \{v_4, u_5, v_3, u_1\}$ 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, $O'_{\alpha=2} = \{v_4, u_5, v_3, u_1, u_4, v_2, v_1, u_2, u_3\}$.*

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 updated 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 6.2 for the overall algorithm.

Theorem 6.2. *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 6.3. *Given one deleted edge (u, v) in graph G, the time complexity of the proposed edge deletion algorithm is $O(f(|\mathsf{CHANGED}|))$, where $f(\cdot)$ is a polynomial function. This also shows that the proposed edge deletion algorithm is bounded.*

7 EXPERIMENTS

7.1 Setup

Table 4: Bipartite graphs used in the experiments.

Graphs	Abbr.	Category	$\|U\|$	$\|V\|$	$\|E\|$
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

Datasets. We utilize ten large bipartite graphs, comprising eight real graphs sourced from KONECT [3], and two synthetic graphs. The synthetic graphs are constructed using a bipartite network model [9], 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.

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

Algorithms. We test the following maintenance algorithms:

- **`Re-compute` [24]:** the state-of-the-art bi-core decomposition algorithm;
- **`BII`* [24]:** the index-based bi-core maintenance algorithm to handle edge insertion;
- **`BIR`* [24]:** the index-based bi-core maintenance algorithm to handle edge deletion;
- **`BNI` [27]:** the state-of-the-art bi-core maintenance algorithm to handle edge insertion;
- **`BNR` [27]:** the state-of-the-art bi-core maintenance algorithm to handle edge deletion;
- **`OI`:** our proposed order-based bi-core maintenance algorithm for handling an edge insertion;
- **`BD`:** our proposed bounded bi-core maintenance algorithm for handling an edge deletion.

We also assess some additional experimental results for extending these algorithms to batch-processing scenarios and testing the space cost of the algorithms. Due to space limitations, the pertinent experimental results are included in our full paper [5].

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 $r \in \{5K, 10K, 15K, 20K, 25K\}$. The edges are randomly selected for all graphs. 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. Note that the default value of r is set to 5,000. 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 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.

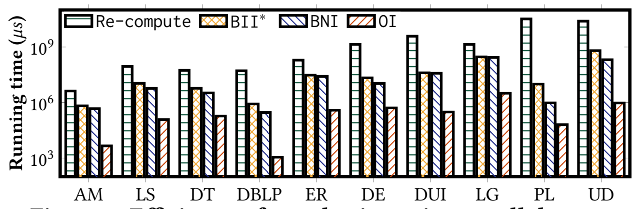

Figure 4: Efficiency of an edge insertion on all datasets.

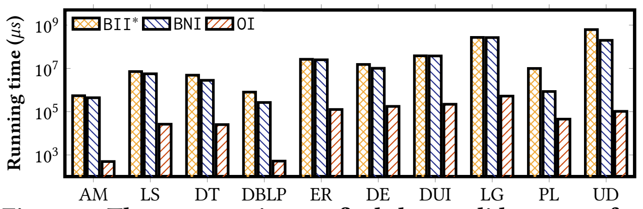

Figure 5: The average time to find the candidate sets after edge insertions.

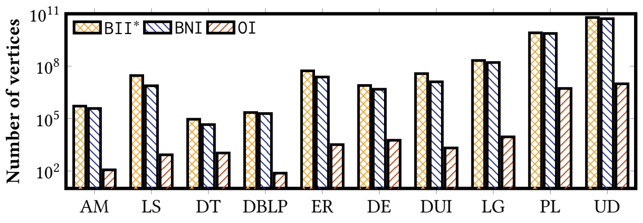

Figure 6: The average number of visited vertices after edge insertions.

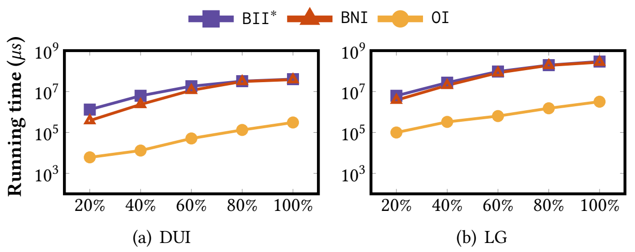

Figure 7: Scalability test for handling edge insertions.

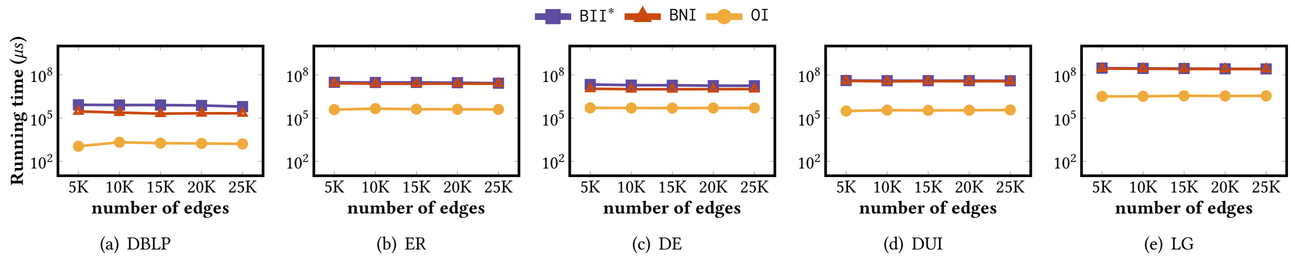

Figure 8: Effect of the number of inserted edges (average time per edge).

• **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 of inserted edges.** Figure 8 depicts the effect of the number of inserted edges on processing time, where $r \in \{5K, 10K, 15K, 20K, 25K\}$. We only present the results on the five largest real-world graphs, because the trends are similar on all other datasets. Note that the running time of `Re-compute` is stable as it needs to decompose the entire graphs, 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. This demonstrates the stability of our proposed algorithm across varying numbers of edge insertions.

• **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 these three algorithms increases with the number of edges, and our algorithm performs better than `BII`* and `BNI` in all cases. Therefore, our proposed edge insertion algorithm scales well on large graphs in practice.

7.3 Efficiency of Edge Deletions

• **Overall efficiency results.** Figure 9 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 of deleted edges.** Figure 11 shows the effect of the number of deleted edges on processing time, where $r \in \{5K, 10K, 15K, 20K, 25K\}$. We only present the results on the five largest real-world graphs, because the trends are similar on all other datasets. Note that we also omit the running time of `Re-compute`

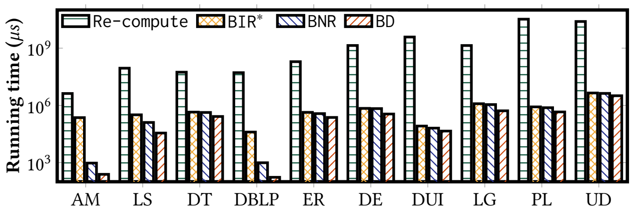

Figure 9: Efficiency of an edge deletion on all datasets.

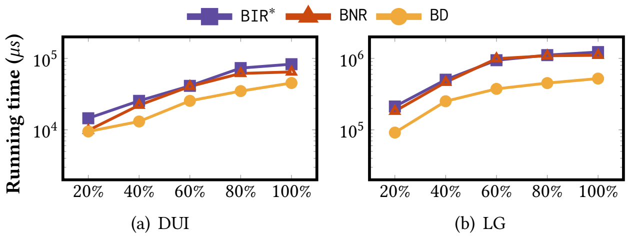

Figure 10: Scalability test for handling edge deletions.

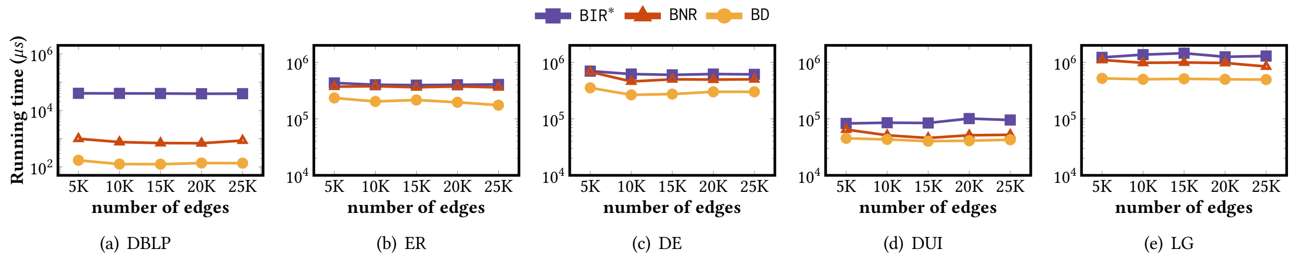

Figure 11: Effect of the number of deleted edges (average time per edge).

here because it is stable. Generally, the findings indicate that with the number of edges increasing, there isn't a substantial rise in the average time it takes to delete each edge. This demonstrates the stability of our proposed algorithm across varying numbers of edge deletions.

• **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 10, 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.

8 RELATED WORKS

Cohesive subgraphs maintenance in unipartite graphs. Identifying cohesive subgraphs is crucial in graph analytics, with common types including k-core [34], k-truss [12], and k-clique [13]. 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, 22, 32, 33, 41, 45], k-trusses [20, 42], and k-cliques [14, 35] 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. [22] and Sariyuce et al. [32] independently noted that the set of affected vertices remains connected after edge insertions or deletions. Leveraging on this observation, [32] proposed an algorithm with linear complexity relative to the size of affected vertices. Conversely, [22] proposed an algorithm with quadratic complexity. Zhang et al. [45] developed a core maintenance algorithm that preserves the k-order between any two vertices, enhancing computational efficiency by effectively identifying the vertices requiring updates. 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 [24, 27]. Liu et al. [24] 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. [27] 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 these algorithms have made notable advancements, they face several limitations. They still involve redundant computations when traversing vertices potentially affected by graph changes via BFS. Moreover, the lack of boundedness analysis on visited vertex counts during this process means there is no theoretical guarantee regarding the relationship between the number of visited vertices and the number of edge insertions or deletions. Therefore, there is significant room for performance improvement in 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 relatively 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.

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] Anonymous Author(s). 2024. An Order-based Approach for Efficient Core Maintenance in Large Bipartite Graphs (full paper). (2024). https://anonymous.4open.science/r/Bicore-Maintenance-Order-Full-Paper-6698/full_paper.pdf

[6] 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.

[7] 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.

[8] 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.

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

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

[11] 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.

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

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

[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] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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).

[29] 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.

[30] 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.

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

[32] 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.

[33] 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.

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

[35] 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.

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

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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 PROPERTIES, LEMMAS, AND THEOREMS

A.1 Proof of Theorem 3.1

Proof. In the bipartite graph G illustrated in Figure 12 (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. □

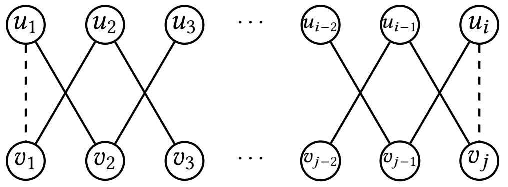

Figure 12: Graph for proving unboundedness.

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 $O(M(G, \Delta G)) = O(|N_1(\text{CHANGED}, G)| \cdot |\text{CHANGED}|)$. Thus, M is a bounded algorithm and the bi-core maintenance is a bounded problem for edge deletions. □

A.3 Proof of Property 2

Proof. Let's first consider setting $\alpha = i$. To establish sufficiency, we utilize the bi-core decomposition algorithm to compute (i, β)-cores. This algorithm operates by removing vertices according to the prescribed order denoted as $O_{\alpha=i}$, ensuring the fulfillment of case (1). In case (2), vertices are removed when they lack sufficient neighbors, as indicated by $0 < lack_{\alpha=i}(w) \leq i$ (for vertices in U) or $0 < lack_{\alpha=i}(w) \leq B_w(i, G) + 1$ (for vertices in V).

For necessity, assume $O_{\alpha=i}$ denotes a BD-Order element for $G_{\alpha=i}$. Since vertices are eliminated in a non-decreasing order based on $B_{(\cdot)}(i, G)$ by the bi-core decomposition algorithm, case (1) is inherently fulfilled. The value $lack_{\alpha=i}(w)$ precisely signifies the disparity between the number of neighbors of w after eliminating all preceding vertices in $O_{\alpha=i}$ and the number required for its inclusion in the $(\alpha, B_w(i,G) + 1)$-core, thereby ensuring the satisfaction of case (2). Similar conclusions apply when considering $\beta = i$. □

A.4 Proof of Lemma 5.1

Proof. For condition (1), suppose the vertex w is in the candidate set with $B_w(i, G) < B_u(i, G)$. The contradiction arises because w being in the candidate set implies it gains new neighbors in the $(i, B_w(i, G) + 1)$-core after the edge insertion. However, since $B_w(i, G) < B_u(i, G)$, both u and v are already in the $(i, B_w(i, G) + 1)$-core originally, leading to no new neighbors for any vertex in that core. Hence, condition (1) holds. For condition (2), $B_w(i, G) > B_v(i, G)$ implies that neither u nor v are part of the original $(i, B_w(i, G))$-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 (3), if $B_u(i, G) \leq B_w(i, G) \leq B_v(i, G)$ and $w \preceq_{\alpha=i} u$, it implies that $B_u(i, G) = B_w(i, G)$, and during the bi-core decomposition, w is removed before u. Therefore, u is already a neighbor for w when w is removed during this process in G. Consequently, even if u becomes a new vertex in the $(i, B_u(i, G) + 1)$-core, the value of $lack_{\alpha=i}(w)$ will remain unchanged. This implies that w cannot be included in the candidate set. For condition (4), if $B_u(i, G) \leq B_w(i, G) \leq B_v(i, G)$ and $u \preceq_{\alpha=i} w$, the value of $lack_{\alpha=i}(w)$ may decrease due to updates in u's c-pairs. These updates initiate with u and propagate locally. Therefore, if there exists a path $w_0, w_1, \cdots, w_t$ such that $w_0 = u$, $w_t = w$, $(w_i, w_{i+1}) \in E$, and $w_i \preceq_{\alpha=i} w_{i+1}$ for $0 \leq i < t$, the decrease in $lack_{\alpha=i}(w_i)$ can propagate locally. As a result, w may be included in the candidate set. □

A.5 Proof of Lemma 5.2

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 $lack_{\alpha=i}(u)$ by one. If $lack_{\alpha=i}(u)$ remains greater than 0 after this decrease, it indicates that u still lacks sufficient neighbors for inclusion in the new core. Therefore, no update can occur. □

A.6 Proof of Lemma 5.3

Proof. For vertex u, after the edge insertion creating a new neighbor v with $u \preceq_{\alpha=i} v$, if an update occurs, we have $B_u(i, G) <$

$B_u(i, G^+)$. Then we prove that $B_u(i, G^+) \leq B_v(i, G) + 1$ and that $B_v(i, G^+)$ can only be $B_v(i, G){+}1$ by contradiction. Assume $B_u(i, G^+) = B_v(i, G){+}k$ ($k \geq 2$). This implies that u must have a new neighbor in the $(i, B_v(i, G) + k)$-core in G^+, which also means that v should be in the $(i, B_v(i, G) + k)$-core in G^+. If we remove u from G^+, all neighbors of u should be contained in a $(i, B_v(i, G) + k - 1)$-core. Hence, v must be in a $(i, B_v(i, G) + k - 1)$-core before the edge insertion. This contradicts with $k \geq 2$. Thus, we can prove that the range for the updated value of u is $B_u(i, G) < B_u(i, G^+) \leq B_v(i, G) + 1$, and the updated value $B_v(i, G^+)$ for v can only be $B_v(i, G) + 1$.

For any vertex w except u and v, we prove that its updated value $B_w(i, G^+)$ can only be $B_w(i, G) + 1$ by contradiction. Assume $B_w(i, G^+) = B_w(i, G){+}k$ ($k \geq 2$). If $w \in U$, at least one neighbor w' of w is newly added to the $(i, B_w(i, G) + k)$-core, not initially in the $(i, B_w(i, G) + 1)$-core. Therefore, w' gains at least k new neighbors in the $(i, B_w(i, G) + k)$-core after the edge insertion. If $w \in V$, there are at least k neighbors of w newly added to the $(i, B_w(i, G) + k)$-core after the edge insertion. Both cases contradict the scenario where initially only one vertex u on the left side and one vertex v on the right side potentially change after inserting (u, v). Hence, the update condition for vertex w except u and v is established. □

A.7 Proof of Lemma 5.4

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 $B_{(\cdot)}(i, G)$ values: (1) If $B_{w'}(i, G) \geq B_w(i, G) + 1$, w''s position in $O'_{\alpha=i}$ remains after w; (2) If $B_{w'}(i, G) = B_w(i, G)$ and w' is also in the candidate set, their positional relationship remains unchanged; (3) If $B_{w'}(i, G) = B_w(i, G)$ 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 $lack_{\alpha=i}(w)$; (4) If $B_{w'}(i, G) < B_w(i, G)$, 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 $lack_{\alpha=i}(w)$ is not less than its value before the update, as demonstrated. □

A.8 Proof of Theorem 5.5

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 Property 2 and Lemma 5.4 post-update. Thus, our algorithm accurately updates c-pairs for all vertices and preserves BD-Order integrity. □

A.9 Proof of Theorem 5.6

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 $\mathsf{AFF}_{(\alpha,\beta)}$ 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 $lack_{\alpha=i}(w)$ 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 5-15 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 $w_i \in V_1$. The vertices in V_2 are identified in Algorithm 2 lines 16-23. 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:

$$\begin{aligned}\textstyle\sum_{w \in V_2} O(d(w)) &\leq \textstyle\sum_{w \in V_2} O(|N_2(\mathsf{AFF}_{(\alpha,\beta)}, G^+)|) \\ &= O(|N_1(\mathsf{AFF}_{(\alpha,\beta)}, G^+)| \cdot |N_2(\mathsf{AFF}_{(\alpha,\beta)}, G^+)|).\end{aligned}$$

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.10 Proof of Lemma 6.1

Proof. If $B_u(\alpha, G) < B_v(\alpha, G)$, v is a neighbor of u in the $(\alpha, B_u(\alpha, G))$-core, but u is not a neighbor of v in the $(\alpha, B_v(\alpha, G))$-core before the edge deletion. Hence, after edge deletion, v is no longer a neighbor of u in the $(\alpha, B_u(\alpha, G))$-core. $sup_{\alpha=i}(u)$ should be decreased by one. If $sup_{\alpha=i}(u) \geq \alpha$ after decreasing, u still has enough neighbors in the $(\alpha, B_u(\alpha, G))$-core, so no update occurs. If $B_u(\alpha, G) > B_v(\alpha, G)$ or $B_u(\alpha, G) = B_v(\alpha, G)$, the proof is similar to that of $B_u(\alpha, G) < B_v(\alpha, G)$, so we omit the proof here. □

A.11 Proof of Theorem 6.3

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 $O(f(|\mathsf{CHANGED}|))$ time. Since the BD-Order and the support value also should be maintained in the algorithm. Given CHANGED, these can be done in $O(f(|\mathsf{CHANGED}|))$ without increasing the time complexity of the deletion algorithm. So, above all, the time complexity of the proposed edge deletion algorithm is $O(f(|\mathsf{CHANGED}|))$, and the edge deletion algorithm is bounded. □

B ADDITIONAL EVALUATIONS

• **Efficiency of batch updates.** Our bi-core maintenance algorithm can be easily adapted for batch updating by following the batch updating framework in [24, 27]. 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 13, our Order-batch algorithm significantly outperforms the methods BI-batch and

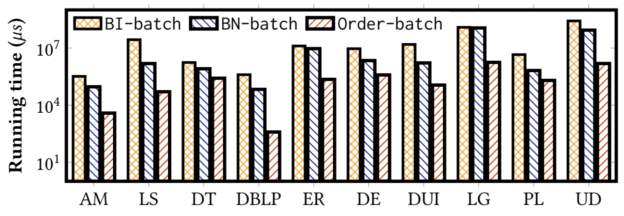

Figure 13: Efficiency of batch update algorithms on all datasets.

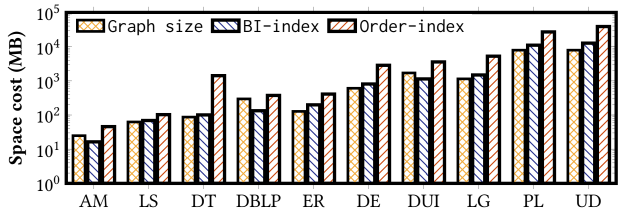

Figure 14: The size of graphs and indexes.

BN-batch, by achieving up to two orders of magnitude in average processing time per edge.

• **Space Cost of Indexes.** Figure 14 shows the memory usage for the graphs, the BI-index from [24], 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.