Accelerating D-core Maintenance over Dynamic Directed Graphs

Xuankun Liao
Hong Kong Baptist University
xkliao@comp.hkbu.edu.hk

Qing Liu
Zhejiang University
qingliucs@zju.edu.cn

Jiaxin Jiang
National University of Singapore
jxjiang@nus.edu.sg

Byron Choi
Hong Kong Baptist University
bchoi@comp.hkbu.edu.hk

Bingsheng He
National University of Singapore
hebs@comp.nus.edu.sg

Jianliang Xu
Hong Kong Baptist University
xujl@comp.hkbu.edu.hk

ABSTRACT

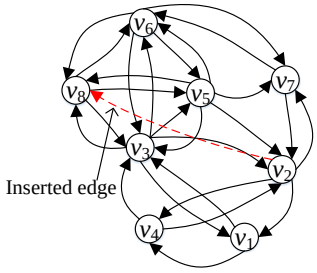
Given a directed graph G and two non-negative integers k and l, a D-core, or (k, l)-core, is the maximal subgraph $H \subseteq G$ where each vertex in H has an in-degree and out-degree not less than k and l, respectively. D-cores have found extensive applications, such as social network analysis, fraud detection, and graph visualization. In these applications, graphs are highly dynamic and frequently updated with the insertions and deletions of vertices and edges, making it costly to recompute the D-cores from scratch to handle the updates. In the literature, the peeling-based algorithm has been proposed to handle D-core maintenance. However, the peeling-based method suffers from efficiency issues, e.g., it may degenerate into recomputing all the D-cores and is inefficient for batch updates. To address these limitations, we introduce novel algorithms for incrementally maintaining D-cores in dynamic graphs. We begin by presenting the theoretical findings to identify the D-cores that should be updated. By leveraging these theoretical analysis results, we propose a local-search-based algorithm with optimizations to handle single-edge insertions and deletions. We further propose an H-index-based algorithm for scenarios involving batch updates. Several novel edge-grouping strategies are proposed to improve the efficiency of the H-index-based algorithm. Extensive empirical evaluations over real-world networks demonstrate that our proposed algorithms are up to 5 orders of magnitude faster than the peeling-based method.

PVLDB Reference Format:
Xuankun Liao, Qing Liu, Jiaxin Jiang, Byron Choi, Bingsheng He, and Jianliang Xu. Accelerating D-core Maintenance over Dynamic Directed Graphs. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/LIAOXuankun/Dynamic-Dcore-public.

1 INTRODUCTION

Directed graphs are omnipresent structures that depict large-scale entities and their relations in various fields, such as *protein-protein interactions* in biological networks [42], *link relationships* in web networks [32], *money flow* in financial networks [29], and *citation relationships* in citation networks [44]. Identifying cohesive subgraphs from directed graphs has attracted considerable attention [29, 32, 38, 48]. One of the well-known cohesive directed graph models is D-core, also known as (k, l)-core, in which every vertex has at least k in-neighbors and l out-neighbors [17]. For example, the directed graph G in Figure 1(a) is a $(2, 2)$-core, as each vertex has at least two in-neighbors and two out-neighbors.



(a) A directed graph G

k \ l	0	1	2	3
0	$v_1v_2v_3v_4$ $v_5v_6v_7v_8$	$v_1v_2v_3v_4$ $v_5v_6v_7v_8$	$v_1v_2v_3v_4$ $v_5v_6v_7v_8$	$v_3v_5v_6v_8$
1	$v_1v_2v_3v_4$ $v_5v_6v_7v_8$	$v_1v_2v_3v_4$ $v_5v_6v_7v_8$	$v_1v_2v_3v_4$ $v_5v_6v_7v_8$	$v_3v_5v_6v_8$
2	$v_1v_2v_3v_4$ $v_5v_6v_7v_8$	$v_1v_2v_3v_4$ $v_5v_6v_7v_8$	$v_1v_2v_3v_4$ $v_5v_6v_7v_8$	$v_3v_5v_6v_8$
3	$v_2v_3v_5v_6$ v_7v_8	$v_2v_3v_5v_6$ v_7v_8	$v_3v_5v_6v_8$	$v_3v_5v_6v_8$

All D-cores of G

k \ l	0	1	2	3
0	$v_1v_2v_3v_4$ $v_5v_6v_7v_8$	$v_1v_2v_3v_4$ $v_5v_6v_7v_8$	$v_1v_2v_3v_4$ $v_5v_6v_7v_8$	$v_3v_5v_6v_8$
1	$v_1v_2v_3v_4$ $v_5v_6v_7v_8$	$v_1v_2v_3v_4$ $v_5v_6v_7v_8$	$v_1v_2v_3v_4$ $v_5v_6v_7v_8$	$v_3v_5v_6v_8$
2	$v_1v_2v_3v_4$ $v_5v_6v_7v_8$	$v_1v_2v_3v_4$ $v_5v_6v_7v_8$	$v_1v_2v_3v_4$ $v_5v_6v_7v_8$	$v_3v_5v_6v_8$
3	$v_2v_3v_5v_6$ v_7v_8	$v_2v_3v_5v_6$ v_7v_8	$v_2v_3v_5v_6$ v_7v_8	$v_3v_5v_6v_8$

All D-cores of $G \oplus (v_2, v_8)$

(b) All D-cores of G and $G \oplus (v_2, v_8)$

v	Anchored coreness
v_1	(0, 2), (1, 2), (2, 2), (2, 2)
v_2	(0, 2), (1, 2), (2, 2), (3, 1)
v_3	(0, 3), (1, 3), (2, 3), (3, 3)
v_4	(0, 2), (1, 2), (2, 2)
v_5	(0, 3), (1, 3), (2, 3), (3, 3)
v_6	(0, 3), (1, 3), (2, 3), (3, 3)
v_7	(0, 2), (1, 2), (2, 2), (3, 1)
v_8	(0, 3), (1, 3), (2, 3), (3, 3)

(c) Vertices' anchored corenesses of G

v	Anchored coreness
v_1	(0, 2), (1, 2), (2, 2), (2, 2)
v_2	(0, 2), (1, 2), (2, 2), (3, 2)
v_3	(0, 3), (1, 3), (2, 3), (3, 3)
v_4	(0, 2), (1, 2), (2, 2)
v_5	(0, 3), (1, 3), (2, 3), (3, 3)
v_6	(0, 3), (1, 3), (2, 3), (3, 3)
v_7	(0, 2), (1, 2), (2, 2), (3, 2)
v_8	(0, 3), (1, 3), (2, 3), (3, 3)

(d) Vertices' anchored corenesses of $G \oplus (v_2, v_8)$

Figure 1: An example of D-core

D-core decomposition is to find all D-cores for a given directed graph [29]. For example, Figure 1(b) shows all D-cores of G and $G \oplus (v_2, v_8)$. Since storing all D-cores can be burdensome, [29] proposed storing the *anchored corenesses* of vertices. Specifically, given a vertex v and a fixed k, we can compute the maximum value of l, denoted by $l_{max}(v, k, G)$, such that v is contained in $(k, l_{max}(v, k, G))$-core. Here, the pair $(k, l_{max}(v, k, G))$ is called an anchored coreness of v. All pairs $(k, l_{max}(v, k, G))$ w.r.t. all k values constitute the anchored corenesses for a vertex v. For example, Figure 1(c) shows the anchored corenesses for each vertex in G.

D-core decomposition has many real-world applications, such as identifying communities of strong cohesiveness in online social networks [8, 13], uncovering hubs and authorities in directed net-

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

works [17], detecting fraudsters in financial networks [23, 28], and analyzing the structure of web graphs [1, 26]. In these applications, directed graphs are highly dynamic with frequent edge and vertex insertions and deletions [48]. For instance, in financial networks, trading activities such as buying and selling stocks happen frequently; in social networks, users frequently engage in interactions; in web graphs, new web pages and links are frequently established. Consequently, the D-cores in these directed graphs may also change, affecting the downstream D-core-related applications. Thus, it is necessary to maintain D-cores for dynamic directed graphs. For example, D-cores can be used to retrieve communities in social networks [8, 13]. Maintaining D-cores can help keep track of up-to-date communities in dynamic directed networks, enhancing the accuracy of friendship recommendations. As another example, by efficiently maintaining D-cores in web graphs, the latest computational results regarding web importance, as well as the updated visualization of web network structures, can be obtained [39].

Fang et al. [13] have proposed a peeling-based method for D-core maintenance. Specifically, given a directed graph G and an edge $e = (u_1, v_1)$ to be inserted or deleted, the peeling-based method first updates $(k, 0)$-cores for all possible k values by iteratively deleting the vertices with the smallest in-degree from G. Then, a threshold value $M = min\{k_{max}(u_1, G), k_{max}(v_1, G)\}$ is computed. Here, $k_{max}(u_1, G)$ denotes the maximal k of vertex u_1 such that a non-empty $(k, 0)$-core contains u_1 in G. Subsequently, for each k value with $0 \leq k \leq M$, the peeling-based method computes the updated (k, l)-cores for all possible values of l from scratch by iteratively deleting the vertices with the minimum out-degree from the $(k, 0)$-core. However, the existing peeling-based method for D-core maintenance suffers from efficiency issues. In more detail, for each k value with $0 \leq k \leq M$, the peeling-based method should compute the updated (k, l)-cores for all possible values of l by employing the peeling algorithm. If M equals the maximum $k_{max}(v, G)$, the peeling-based method will degenerate into the D-core decomposition method, which is time-consuming. Furthermore, for batch updates, the peeling-based method handles the updated edges one by one. As the batch size increases, the D-core maintenance time deteriorates, making it even worse than the performance of D-core decomposition algorithm.

Motivated by the above observations, in this paper, we propose novel algorithms to accelerate D-core maintenance. To this end, first, we conduct a comprehensive analysis of how edge insertions and deletions affect the anchored corenesses of vertices. A series of theorems is devised to enable efficient identification of vertices requiring anchored coreness updates. For example, given an updated edge (u_1, v_1) for directed graph G, assuming that $k_{max}(v_1, G) = k \leq k_{max}(u_1, G)$, we find that only vertices w with $k_{max}(w, G)$ equal to k and reachable from v_1 through a set of vertices z with $k_{max}(z, G)$ of k *may* have $k_{max}(w, G)$ updated, and $|k_{max}(w, G') - k_{max}(w, G)| \leq 1$ holds. Here, $E_{G'} = E_G \cup \{(u_1, v_1)\}$ and $V_{G'} = V_G$.[1] A similar conclusion holds for $l_{max}(v, k, G)$ of each vertex v.

Based on the theoretical findings, we first introduce a two-phase local-search-based algorithm for D-core maintenance with single-edge updates. In the first phase, the algorithm maintains $k_{max}(v, G)$ for each $v \in V_G$, based on which it concurrently updates $l_{max}(v, k, G)$ for $0 \leq k \leq min\{k_{max}(u_1, G'), k_{max}(v_1, G')\}$ in the second phase. Here, (u_1, v_1) is the updated edge. In each phase, the algorithm performs a local search starting from the updated edge to collect candidate vertices that may require anchored coreness updates based on the proposed theorems. Subsequently, vertices that do not meet certain degree constraints are iteratively removed from the candidates. Finally, the remaining candidates w will have their $k_{max}(w, G)$ or $l_{max}(w, k, G)$ updated. Moreover, a series of optimizations, including avoiding the processing of identical $(k, 0)$-cores, skipping redundant $l_{max}(v, k, G)$ maintenance, and reusing previous maintenance results, are proposed to enhance the performance.

To improve the efficiency of batch updates, we propose an H-index-based algorithm that locally computes the anchored corenesses for each vertex. The H-index-based algorithm also consists of two stages. In each stage, the algorithm utilizes the concept of H-index [20] to iteratively calculate $k_{max}(v, G)$ or $l_{max}(v, k, G)$ for each $v \in V_G$ in parallel. To further improve the efficiency of batch updates, we introduce several edge grouping strategies to identify edges that can be handled at the same time. By simultaneously inserting or deleting the grouped edges, we can precisely initialize the vertex values close to the updated ones, thereby improving the efficiency of batch updates.

We summarize the main contributions of this paper as follows:

- For the first time in the literature, we systemically study the problem of D-core maintenance and develop theorems to identify vertices that need to be visited to accomplish D-core maintenance.
- We propose a local-search-based algorithm to accelerate D-core maintenance with single-edge updates. Three optimizations are introduced to further improve performance.
- We design an efficient H-index-based maintenance algorithm to handle D-core maintenance with batch updates. In addition, two edge grouping strategies are proposed to enhance efficiency.
- Both theoretical analysis and empirical evaluation validate the efficiency of our algorithms for D-core maintenance.

The rest of this paper is organized as follows. Section 2 reviews related works. Section 3 presents the formal definition of the studied problem. Section 4 presents the theoretical basis of this paper. Sections 5 and 6 propose the local-search-based algorithm and the H-index-based algorithm, respectively. Experimental results are reported in Section 7. Finally, Section 8 concludes the paper.

2 RELATED WORK

In this section, we review the related work from two aspects, i.e., *core decomposition* and *core maintenance*.

Core Decomposition. A k-core is the maximal subgraph of an undirected graph where each vertex has at least k neighbors within the subgraph [47]. Core decomposition aims to compute each vertex's coreness, which is the maximal value of k for a vertex to be contained in a non-empty k-core [39]. Numerous efficient algorithms have been devised to address core decomposition in undirected graphs, including peeling-based approaches [6, 9, 25], disk-based methods [9, 25], semi-external methods [50], parallel algorithms [10, 12], and distributed solutions [3, 40, 41].

[1] Throughout this paper, we consistently use G to denote the original graph and G' to represent the new graph updated from G.

Moreover, core decomposition has been studied for various types of graphs, including weighted graphs [11], uncertain graphs [7, 43], bipartite graphs [31], temporal graphs [15, 51], and heterogeneous information networks [14]. Additionally, previous work has explored the core decomposition problem in directed graphs based on the D-core model [17], and efficient peeling-based algorithms and distributed algorithms for D-core decomposition have been developed [13, 29]. Note that the core decomposition algorithms are mainly designed for static graphs, and thus are not efficient for the core maintenance over dynamic graphs.

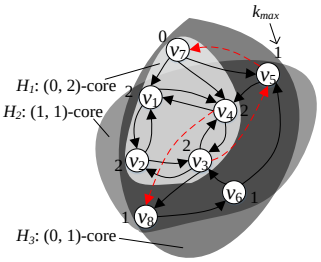
Core Maintenance. When a graph undergoes changes, the problem of core maintenance is to update the coreness of vertices. Sariyuce et al. [45, 46] demonstrated that the insertion/deletion of a single edge can cause a vertex's coreness to change by at most one, based on which efficient incremental core maintenance algorithms are proposed. Similar conclusions have also been presented in [27]. Building upon these findings, various parallel techniques that concurrently process batches of updated edges for core maintenance have been proposed [5, 21, 24, 49]. Liu et al. [34] introduced an approximate algorithm to handle core maintenance with batch updates. Moreover, a distributed core maintenance algorithm has been proposed in [2]. [18, 19, 52] employed the k-order to facilitate the core maintenance. Besides, the core maintenance problem has been studied for different types of core models, including distance-generalized core [33], colorful h-star core [16], hierarchical core [30], hypercore [22, 35, 36], and (α, β)-core [4, 31, 37].

In summary, (i) most previous works on core maintenance fail to consider the directionality of edges and, therefore, are not applicable to D-core maintenance over directed graphs; (ii) the peeling-based method may degrade into the D-core decomposition method and sequentially handles updated edges for batch updates, thereby leading to efficiency issues. Consequently, more efficient algorithms for D-core maintenance over dynamic directed graphs are urgently needed.

3 PROBLEM FORMULATION

We consider a directed, unweighted simple graph $G = (V_G, E_G)$, where V_G and E_G represent the sets of vertices and edges, respectively. For brevity, we refer to a directed graph as a `digraph`. Each directed edge $e = (u, v) \in E_G$ represents a connection from vertex u to vertex v. If the edge (u, v) exists, u is an in-neighbor of v, and v is an out-neighbor of u. For a vertex v, we denote all of its in-neighbors and out-neighbors in G by $N^+_G(v) = \{u : (u, v) \in E_G\}$ and $N^-_G(v) = \{u : (v, u) \in E_G\}$, respectively. The neighbors of vertex v is defined as $N_G(v) = N^+_G(v) \cup N^-_G(v)$. Correspondingly, three types of vertex degree are defined as follows: (1) v's in-degree is the number of v's in-neighbors in G, i.e., $deg^{in}_G(v) = |N^+_G(v)|$; (2) v's out-degree is the number of v's out-neighbors in G, i.e., $deg^{out}_G(v) = |N^-_G(v)|$; (3) v's degree is the sum of its in-degree and out-degree, i.e., $deg_G(v) = deg^{in}_G(v) + deg^{out}_G(v)$. Based on the in-degree and out-degree, we introduce the definition of D-core as follows.

Definition 3.1. ***(D-core [17]).*** *Given a digraph $G = (V_G, E_G)$ and two integers k and l, a D-core of G, also denoted as (k, l)-core, is the maximal subgraph $H = (V_H, E_H) \subseteq G$ such that $\forall v \in V_H$, $deg^{in}_H(v) \geq k$ and $deg^{out}_H(v) \geq l$.*



(a) A directed graph G

Before update		After update	
v	Anchored coreness	v	Anchored coreness
v_1	(0, 2), (1, 2), (2, 2)	v_1	(0, 2), (1, 2), (2, 2)
v_2	(0, 2), (1, 2), (2, 2)	v_2	(0, 2), (1, 2), (2, 2)
v_3	(0, 2), (1, 2), (2, 2)	v_3	(0, 2), (1, 2), (2, 2)
v_4	(0, 2), (1, 2), (2, 2)	v_4	(0, 2), (1, 2), (2, 2)
v_5	(0, 1), (1, 1)	v_5	(0, 2), (1, 2)
v_6	(0, 1), (1, 1)	v_6	(0, 1), (1, 1)
v_7	(0, 2)	v_7	(0, 2), (1, 2)
v_8	(0, 1), (1, 1)	v_8	(0, 1), (1, 1), (2, 0)

(b) Anchored coreness update

Figure 2: A digraph G with batch insertion (inserted edges and updated anchored corenesses are highlighted in red)

According to Definition 3.1, a D-core should satisfy both the degree constraints and the size constraint. The degree constraints ensure the cohesiveness of D-core in terms of in-degree and out-degree. The size constraint guarantees the uniqueness of the D-core, i.e., for a specific pair of (k, l), at most one D-core exists in G. Additionally, D-core has a *partial nesting* property as follows.

Property 3.1. ***Partial Nesting [29].*** *Given two D-cores, (k_1, l_1)-core D_1 and (k_2, l_2)-core D_2, D_1 is nested in D_2 (i.e., $D_1 \subseteq D_2$) if $k_1 \geq k_2$ and $l_1 \geq l_2$.*

Consider the digraph G before the update, as shown in Figure 2. The subgraph H_1 induced by vertices v_1, v_2, v_3, v_4, and v_7 is a $(0, 2)$-core, as each vertex in H_1 has in-degree and out-degree no less than 0 and 2, respectively. In addition, we can observe that $H_1 \subseteq H_3 = (0, 1)$-core and $H_1 \nsubseteq H_2 = (1, 1)$-core. According to the partial nesting property, if a vertex v is in a (k, l)-core, v is also within all (k, l')-cores with $l' < l$. Therefore, we only need to keep the maximum l to record all the D-cores containing v under the fixed k for v. To this end, the anchored coreness is defined as follows.

Definition 3.2. ***(Anchored Coreness [29]).*** *Given a digraph G and an integer k, the anchored coreness of a vertex $v \in V_G$ w.r.t. k is a pair $(k, l_{max}(v, k, G))$, where $l_{max}(v, k, G) = \max_{l \in \mathbb{N}_0} \{l \mid \exists (k, l)\text{-core } H \subseteq G \wedge v \in V_H\}$. The* entire anchored corenesses *of the vertex v are defined as $\Phi(v) = \{(k', l_{max}(v, k', G)) \mid 0 \leq k' \leq k_{max}(v, G)\}$, where $k_{max}(v, G) = \max_{k'' \in \mathbb{N}_0} \{k'' \mid \exists (k'', 0)\text{-core } H \subseteq G \wedge v \in V_H\}$.*

We omit G in $k_{max}(v, G)$ and $l_{max}(v, k, G)$ when the context is clear. Besides, we use k_{max} ($l_{max}(k)$) and $k_{max}(v, G)$ ($l_{max}(v, k, G)$) interchangeably. We denote the set of anchored coreness $\Phi(v)$ for each vertex $v \in V_G$ as $\Phi(G)$, and the maximal $k_{max}(v, G)$ in a graph G as k^G_{max}, i.e., $\Phi(G) = \{\Phi(v) \mid \forall v \in V_G\}$ and $k^G_{max} = max\{k_{max}(v, G) \mid \forall v \in V_G\}$. The anchored coreness is two-dimensional, which simultaneously takes into account the in-degree and out-degree of vertices in directed graphs. Take the graph G before the update in Figure 1(a) as an example. Let $k = 3$; the anchored coreness of vertex v_8 is $(3, 3)$, as $l_{max}(v_8, 3, G) = 3$. In addition, since $k_{max}(v_8, G) = 3$, $\Phi(v_8) = \{(0, 3), (1, 3), (2, 3), (3, 3)\}$. Based on the above concepts, the D-core decomposition and maintenance are defined as follows.

Definition 3.3. ***(D-core Decomposition [29]).*** *Given a digraph G, D-core decomposition is to compute the anchored corenesses $\Phi(v)$ for each vertex v in G.*

Problem 1. **(D-core Maintenance).** *Given a digraph $G = (V_G, E_G)$ and a batch of edges E_i to be inserted/deleted, the problem of D-core maintenance is to update the anchored corenesses $\Phi(v)$ for each vertex in the updated digraph G', where $E_{G'} = E_G \cup E_i$ for the edge insertion case and $E_{G'} = E_G \setminus E_i$ for the edge deletion case, and $V_{G'} = V_G$.*

Example 3.1. *Consider the digraph G in Figure 2. After inserting edges $E_i = \{(v_5, v_0), (v_4, v_8), (v_3, v_5)\}$, the anchored corenesses of vertices in G are updated. The updated anchored corenesses are highlighted in red in Figure 2(b).*

It is noteworthy that since vertex insertion and deletion can be simulated by a sequence of edge insertions and deletions [33], we focus on edge insertions/deletions in this paper.

4 THEORETICAL BASIS

In this section, we present detailed theoretical analysis that forms the foundation of our new algorithms. Specifically, we first introduce novel definitions and theorems to identify vertices that necessitate updating k_{max} due to a single-edge update. These theoretical findings can be readily extended to the analysis of $l_{max}(k)$ by solely considering vertices in the $(k,0)'$-core. Here, $(k,0)'$-core is the updated $(k,0)$-core. Next, we show theorems that illustrate the update of different $(k,0)$-cores after maintaining $k_{max}(v,G)$ for each $v \in V_G$. Once the inserted or removed edges from the $(k,0)$-core are identified, the maintenance of $l_{max}(w,k)$ for $w \in V_{(k,0)'-\text{core}}$ can be easily accomplished using a similar approach to the maintenance of $k_{max}(v,G)$ based on the extended theoretical findings. Note that due to space limitations, the proofs of all the theorems in this paper are omitted and can be found in the technical report.

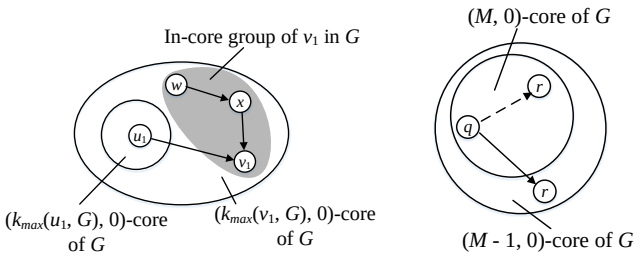
Theorem 4.1. *Given a digraph $G = (V_G, E_G)$ and an inserted or deleted edge (u_1, v_1), $k_{max}(v,G)$ of any vertex $v \in V_G$ can change by at most 1.*

Theorem 4.2. *Given an inserted/deleted edge (u_1, v_1) of $G = (V_G, E_G)$, if $k_{max}(u_1,G) \geq k_{max}(v_1,G)$, then $k_{max}(v_1,G)$ may be updated. Otherwise, no vertex $v \in V_G$ will change $k_{max}(v,G)$.*

Theorem 4.2 demonstrates that given an updated edge (u_1, v_1), the endpoint v_1 with smaller $k_{max}(v_1,G)$ *and* having $deg^{in}_G(v_1)$ changed may have $k_{max}(v_1,G)$ updated. Here, the condition of $k_{max}(u_1,G) \geq k_{max}(v_1,G)$, and the degree condition, i.e., the edge must be from u_1 to v_1, must be satisfied simultaneously. Otherwise, no vertex v in G will have $k_{max}(v,G)$ updated. Figure 3(a) shows an example where $k_{max}(v_1,G)$ may be updated if (u_1, v_1) is inserted/deleted. Additionally, each vertex v in G' can have its $k_{max}(v,G)$ changed by at most 1 by Theorem 4.1.

Definition 4.1. ***(In-Core Group).*** *Given a digraph $G = (V_G, E_G)$ and a vertex $v \in V_G$, the in-core group of v is the maximal connected subgraph $H \subseteq G$ where each vertex $w \in V_H$ satisfies $k_{max}(w,G) = k_{max}(v,G)$ and is reachable from v through a set of vertices x that have $k_{max}(x,G)$ equal to $k_{max}(v,G)$.*

We denote the in-core group of v as $C^{in}(v)$. The notation of in-core group is motivated by the subcore proposed in [45]. However, our definition differs from subcore in that we use $k_{max}(v,G)$ of each vertex $v \in V_G$ to determine whether vertices are in the same in-core group rather than the one-dimensional coreness used in [45]. For example, $C^{in}(v_2)$ in Figure 1(a) without considering the inserted edge includes vertices $\{v_2, v_3, v_5, v_6, v_7, v_8\}$, as they all have $k_{max}(v,G)$ of 3 and are connected to v_2 via vertices with $k_{max}(v,G)$ of 3.



(a) Illustrating Theorems 4.2 and 4.3 (b) Illustrating Theorem 4.6

Figure 3: Simple illustrative example for our theorems ((q,r) will be inserted into $(M,0)$-core with $k_{max}(r,G)$ incremented from $M-1$ to M in Figure 3(b))

Theorem 4.3. *Given an inserted or deleted edge (u_1, v_1) of $G = (V_G, E_G)$, assuming $k_{max}(u_1,G) \geq k_{max}(v_1,G)$ and $k_{max}(v_1,G)$ is updated, then only vertices v in $C^{in}(v_1)$ may have $k_{max}(v,G)$ changed.*

Besides the vertex v_1 mentioned in Theorem 4.2, Theorem 4.3 reveals that vertices $v \in C^{in}(v_1)$ may also have $k_{max}(v,G)$ changed with single-edge updates. For example, in Figure 3(a), vertices w and x in the in-core group of v_1 may have $k_{max}(w,G)$ and $k_{max}(x,G)$ updated if (u_1, v_1) is updated. Inspired by the notations of MCD and PCD in [45], we then introduce two useful definitions and corresponding theorems to help us prune the search space retrieved by Theorem 4.3.

Definition 4.2. ***(Exceptional Degree).*** *Given a digraph $G = (V_G, E_G)$, for each vertex $v \in V_G$, we say that u is an exceptional in-neighbor of v if $\{(u,v)\} \cap E_G \neq \emptyset$ and $k_{max}(u,G) \geq k_{max}(v,G)$. The number of exceptional in-neighbors of v is termed as the exceptional degree of v, denoted as $ED(v,G)$.*

Definition 4.3. ***(Pure Exceptional Degree).*** *The pure exception degree of a vertex v, denoted by $PED(v,G)$, is the number of in-neighbor w of v, such that $k_{max}(w,G) > k_{max}(v,G)$ **or** $k_{max}(w,G) = k_{max}(v,G)$ and $ED(w,G) > k_{max}(v,G)$.*

The exceptional degree counts the number of in-neighbors u of v with $k_{max}(u,G)$ no less than $k_{max}(v,G)$, which is also the number of in-neighbors of v in the $(k_{max}(v,G),0)$-core. The pure exceptional degree is stricter than the exceptional degree, counting two types of in-neighbors of v. The first type is those in-neighbors u with $k_{max}(u,G)$ larger than $k_{max}(v,G)$, and the second type is those in-neighbors w having the same $k_{max}(w,G)$ as $k_{max}(v,G)$ but with enough in-neighbors that may cause $k_{max}(w,G)$ increase (i.e., $ED(w,G) > k_{max}(v,G)$). $PED(v,G)$ indicates v's potential in-degree in the $(k_{max}(v,G'),0)$-core with a single-edge insertion. We then derive two theorems to illustrate how $ED(v,G')$ and $PED(v,G')$ can help prune vertices v that will not have $k_{max}(v,G)$ changed.

Theorem 4.4. *Given a digraph G and a deleted edge (u_1, v_1),*

Algorithm 1: Local-search-based D-core maintenance algorithm for a single-edge insertion

Input: digraph $G = (V_G, E_G)$, inserted edge (u_1, v_1), original anchored corenesses $\Phi(G)$
Output: updated anchored corenesses $\Phi(G')$

1 Let G' be a graph with $V_{G'} = V_G$ and $E_{G'} = E_G \cup \{(u_1, v_1)\}$;
2 Update $k_{max}(v, G)$ for each vertex $v \in V_{G'}$ using Algorithm 2;
3 Update $l_{max}(v, k, G)$ for each vertex $v \in V_{G'}$ by invoking Algorithm 3 with $k \in [0, min\{k_{max}(u_1, G'), k_{max}(v_1, G')\}]$;
4 **return** updated anchored corenesses of G' as $\Phi(G')$;

for a vertex $v \in V_G$, if $ED(v, G') < k_{max}(v, G)$, then v will have $k_{max}(v, G)$ decremented.

Theorem 4.5. *Given a digraph G and an inserted edge (u_1, v_1), for a vertex $v \in V_G$, if $PED(v, G') \leq k_{max}(v, G)$, then v will not have $k_{max}(v, G)$ value incremented.*

Theorems 4.4 and 4.5 help to prune vertex v by comparing $k_{max}(v, G)$ with $ED(v, G')$ and $PED(v, G')$. We then present the theorems that reveal how the structure of $(k, 0)$-cores with different k of G will be updated with $\{k_{max}(v, G') : \forall v \in V_{G'}\}$. Based on Theorems 4.2 and 4.3, given a digraph G and an inserted/deleted edge (u_1, v_1), let $M = min\{k_{max}(u_1, G'), k_{max}(v_1, G')\}$,[2] then the update of (u_1, v_1) has no effect on $(k, 0)$-cores with $k \geq M + 1$ [13]. Next, we present how the $(k, 0)$-cores with $k < M + 1$ change.

Theorem 4.6. *Given a digraph G and an inserted edge (u_1, v_1) with $M = min\{k_{max}(u_1, G'), k_{max}(v_1, G')\}$, where G' is a digraph with $E_{G'} = E_G \cup \{(u_1, v_1)\}$ and $V_{G'} = V_G$. Let G_k denote the original $(k, 0)$-core in G and G'_k denote the updated $(k, 0)$-core. Then:*

(i) *$E_{G'_k} = E_{G_k} \cup \{(u_1, v_1)\}$ and $V_{G'_k} = V_{G_k}$, where $0 \leq k \leq M-1$.*
(ii) *if some vertices $V_w \in V_{G'}$ have their k_{max} incremented, then $E_{G'_M} = E_{G_M} \cup \{(u_1, v_1)\} \cup E_{V_w}$ and $V_{G'_M} = V_{G_M} \cup V_w$. Here, for any $e = (q, r) \in E_{V_w}$, $\{k_{max}(r, G) < k_{max}(r, G') = M \wedge r \in V_w\} \vee \{k_{max}(q, G) < k_{max}(q, G') = M \wedge q \in V_w\}$ holds. Otherwise, $E_{G'_M} = E_{G_M} \cup \{(u_1, v_1)\}$ and $V_{G'_M} = V_{G_M}$.*

Theorem 4.6 demonstrates how the structure of $(k, 0)$-cores with $k < M+1$ is updated after performing k_{max} maintenance with the insertion of (u_1, v_1), forming the foundation for maintaining $l_{max}(k)$ after k_{max} maintenance. Considering the example in Figure 3(b), where $k_{max}(r, G) = M - 1$. When $k_{max}(r, G)$ is incremented to M, (q, r) will be inserted into the $(M, 0)$-core in G'. Since the theorem for the edge deletion case can be extended trivially, we omit the details for simplicity.

Summary. This section has shown that given an updated edge (u_1, v_1) of digraph G with $k_{max}(u_1, G) \geq k_{max}(v_1, G)$, only vertices v in the in-core group $C(v_1)$ with pure exceptional degree $PED(v, G') > k_{max}(v, G)$ ($ED(v, G') \geq k_{max}(v, G)$ for edge deletion) may have $k_{max}(v, G)$ incremented (decremented) by 1. Besides, we demonstrate that only (u_1, v_1) will be inserted into (removed from) $(k, 0)$-cores with $k < M$, and multiple edges may be inserted into (removed from) $(M, 0)$-core. $(k, 0)$-cores with $k \geq M + 1$ will not be affected. Here, $M = min\{k_{max}(u_1, G'), k_{max}(v_1, G')\}$.

[2]It is stated in [13] that $M = max\{k_{max}(u_1, G'), k_{max}(v_1, G')\}$ for the edge deletion case, we have corrected this statement here in this paper.

Algorithm 2: Updating $k_{max}(v, G)$ for each vertex $v \in V_{G'}$ with a single-edge insertion

Input: digraph $G' = (V_{G'}, E_{G'})$, inserted edge (u_1, v_1), original anchored corenesses $\Phi(G)$
Output: updated $k_{max}(v, G)$ $\mathcal{K} = \{k_{max}(v, G') : \forall v \in V_{G'}\}$

1 Let x be an integer and $x \leftarrow min\{k_{max}(u_1, G), k_{max}(v_1, G)\}$;
2 Let S be an empty stack and $S \leftarrow \emptyset$;
3 Let r be a vertex and $r \leftarrow v_1$;
4 **if** $k_{max}(u_1, G) < k_{max}(v_1, G)$ **then**
5 **return** $\mathcal{K} = \{k_{max}(v, G) : \forall v \in V_{G'}\}$;
6 **for** $v \in V_{G'}$ **do**
7 visited[v] $\leftarrow$ *false*; deleted[v] $\leftarrow$ *false*; d[v] $\leftarrow 0$;
8 Calculate $ED(v, G')$ and $PED(v, G')$;
9 d[r] $\leftarrow PED(r, G')$; S.push(r); visited[r] $\leftarrow$ *true*;
10 **while** $Q \neq \emptyset$ **do**
11 $v \leftarrow S$.pop();
12 **if** d[v] $> x$ **then**
13 **for** $(v, w) \in E_{G'}$ **do**
14 **if** $k_{max}(w, G) = x$ ***and*** $ED(w, G') > x$ ***and*** visited[w] *is false* **then**
15 S.push(w); visited[w] $\leftarrow$ *true*;
16 d[w] $\leftarrow$ d[w] + $PED(w, G')$;
17 **else**
18 **if** deleted[v] *is false* **then**
19 REMOVE ($G' = (V_{G'}, E_{G'})$, $\mathcal{K}$, d, deleted, x, v);
20 **for** $v \in V_{G'}$ **do**
21 **if** deleted[v] *is false* ***and*** visited[v] *is true* **then**
22 $k_{max}(v, G') \leftarrow k_{max}(v, G) + 1$;
23 **return** updated $k_{max}(v, G)$ $\mathcal{K} = \{k_{max}(v, G') : \forall v \in V_{G'}\}$;

Function REMOVE

Input: $G = (V_G, E_G)$, $\mathcal{K}$, d, deleted, x, v

1 deleted[v] $\leftarrow$ *true*;
2 **for** $(v, w) \in E_G$ **do**
3 **if** $k_{max}(w, G) = x$ **then**
4 d[w] $\leftarrow$ d[w] - 1;
5 **if** d[w] $= x$ ***and*** deleted[w] *is false* **then**
6 REMOVE ($G = (V_G, E_G)$, $\mathcal{K}$, d, deleted, x, w)

5 ALGORITHMS FOR SINGLE-EDGE UPDATES

In this section, we propose a local-search-based incremental algorithm for D-core maintenance with single-edge updates. The algorithms for the edge insertion and deletion are presented in Sections 5.1 and 5.2, respectively. Then, we propose several non-trivial optimizations to accelerate the algorithms in Section 5.3.

5.1 Local-search-based D-core Maintenance Algorithm for a Single-edge Insertion

In this section, we present a local-search-based D-core maintenance algorithm for a single-edge insertion.

Overview. Given an inserted edge (u_1, v_1) and a digraph G, $\forall v \in V_G$, the general idea is first to maintain $k_{max}(v)$ and then update $l_{max}(v, k)$ with $k \in [0, min\{k_{max}(u_1, G'), k_{max}(v_1, G')\}]$. The algorithm framework is outlined in Algorithm 1, which consists of two phases: 1) updating $k_{max}(v)$; 2) maintaining $l_{max}(v, k)$. Main-

Algorithm 3: Updating $l_{max}(v,k,G)$ for each vertex $v \in V_{G'}$ with a single-edge insertion

Input: digraph $G' = (V_{G'}, E_{G'})$, inserted edge (u_1, v_1), updated $k_{max}(v,G)$ $\mathcal{K} = \{k_{max}(v,G') : \forall v \in V_{G'}\}$, original anchored corenesses $\Phi(G)$
Output: updated anchored corenesses $\Phi(G')$

1 Let $M \leftarrow min\{k_{max}(u_1,G'), k_{max}(v_1,G')\}$;
2 **for** $i \leftarrow 0$ *to* M **do**
3 Let G_i be the $(k,0)$-core in G;
4 Let G_i' be the updated $(k,0)$-core in G';
5 **for** $k \leftarrow 0$ *to* $M-1$ **do in parallel**
6 Insert (u_1,v_1) into G_k and maintain $l_{max}(v,k,G)$ for each $v \in V_{G_k'}$;
7 **if** $k_{max}(u_1,G) \geq k_{max}(v_1,G)$ **then**
8 $\mathcal{E} \leftarrow E_{G_M'} \setminus E_{G_M}$;
9 **else**
10 $\mathcal{E} \leftarrow \{(u_1,v_1)\}$;
11 Maintain $l_{max}(v,M,G)$ for $v \in V_{G_M'}$ by inserting $e \in \mathcal{E}$;
12 **return** the updated anchored corenesses as $\Phi(G')$;

taining $k_{max}(v)$ and $l_{max}(v,k')$ with a specific k' shares the same core idea: first locating a set of candidate vertices using local search and then iteratively removing those vertices that do not meet specific degree constraints from the candidates. Finally, the remaining candidates w will have their $k_{max}(w,G)$ and $l_{max}(w,k')$ updated.

Phase I: Maintaining $k_{max}(v,G)$. How to maintain k_{max} of vertices is outlined in Algorithm 2. It starts with initialization and calculation of the $ED(v,G')$ and $PED(v,G')$ for each $v \in V_{G'}$ (lines 1-8), during which the $\mathsf{d}[v]$ (potential in-degree of v in $(x,0)$-core of G') will be initialized to 0 as well. If $k_{max}(u_1) < k_{max}(v_1)$, no vertices v in G' will have $k_{max}(v)$ updated based on Theorem 4.2. Hence, Algorithm 2 will stop the maintenance (line 4-5). Otherwise, a DFS-based search is ignited from r (line 9), and the vertex v at the top of the stack will be checked (line 11). If v and its unvisited out-neighbor w have the potential to have $k_{max}(v)$ and $k_{max}(w)$ incremented based on Theorems 4.5 and 4.3, w will be put into the stack for further check (lines 12-16). If v cannot be in $(k_{max}(v,G)+1,0)$-core based on Theorem 4.5, a recursive deletion will be started from v, during which all the out-neighbors w of v with $k_{max}(w,G)$ equaling to x will have $\mathsf{d}[w]$ decremented by 1 (lines 17-19). If $\mathsf{d}[w]$ is x after the decrement and w is not marked as deleted, a new deletion will be started from w (lines 4-6 of the Function REMOVE). Finally, all the visited but not yet deleted vertices v will have $k_{max}(v,G)$ incremented by 1 (lines 20-22).

Example 5.1. *We use the digraph G in Figure 1 to illustrate Algorithm 2. The update of D-cores in G before and after the insertion of $\{(v_2,v_8)\}$ are shown in Figure 1(b). The algorithm first chooses vertex v_8 as r and calculates $ED(v,G')$ and $PED(v,G')$ for each vertex $v \in V_{G'}$. Since v_8 has only one in-neighbor v_6 with $k_{max}(v_6) = k_{max}(v_8) = 3$ and $ED(v_6) = 4 > 3$, $PED(v_8)$ equals to 1, which is smaller than $x = 3$. Hence, the algorithm removes v_8 with Function REMOVE. Since no vertices are added to Q after the removal of v_8, the algorithm continues to transverse all vertices in $V_{G'}$, and no vertex v will have $k_{max}(v)$ incremented.*

Phase II: Maintaining $l_{max}(v,k,G)$. With obtained $\{k_{max}(v,G') : \forall v \in V_{G'}\}$, how $l_{max}(k)$ are maintained is presented in Algorithm 3.

Algorithm 4: Updating $k_{max}(v,G)$ for each vertex $v \in V_{G'}$ with a single-edge deletion

Input: digraph $G = (V_G, E_G)$, deleted edge (u_1,v_1), original anchored corenesses $\Phi(G)$
Output: updated $k_{max}(v,G)$ $\mathcal{K} = \{k_{max}(v,G') : \forall v \in V_{G'}\}$

1 Let G' be a graph with $V_{G'} = V_G$ and $E_{G'} = E_G \setminus \{(u_1,v_1)\}$;
2 Let x be an integer and $x \leftarrow min\{k_{max}(u_1,G), k_{max}(v_1,G)\}$;
3 Let r be a vertex and $r \leftarrow v_1$;
4 **if** $k_{max}(u_1,G) < k_{max}(v_1,G)$ **then**
5 **return** $\mathcal{K} = \{k_{max}(v,G) : \forall v \in V_{G'}\}$;
6 **if** $k_{max}(u_1) \neq k_{max}(v_1)$ **then**
7 $\mathsf{d}, H \leftarrow$ FINDINCORE $(G', \mathcal{K}, r)$;
8 **else**
9 $\mathsf{d1}, H_1 \leftarrow$ FINDINCORE $(G', \mathcal{K}, u_1)$;
10 $\mathsf{d2}, H_2 \leftarrow$ FINDINCORE $(G', \mathcal{K}, v_1)$;
11 $\mathsf{d} \leftarrow \mathsf{d1} \cup \mathsf{d1}$; $H \leftarrow H_1 \cup H_2$;
12 Sort vertices in H in increasing order w.r.t. d;
13 **for** $v \in V_H$ **do**
14 **if** $\mathsf{d}[v] < M$ **then**
15 $k_{max}(v,G') \leftarrow k_{max}(v,G) - 1$;
16 **for** $(v,w) \in E_H$ **do**
17 **if** $\mathsf{d}[w] > \mathsf{d}[v]$ **then**
18 $\mathsf{d}[w] \leftarrow \mathsf{d}[w] - 1$;
19 Resort vertex in V_H;
20 **else**
21 **Break**;
22 **return** updated $k_{max}(v,G)$ $\mathcal{K} = \{k_{max}(v,G') : \forall v \in V_{G'}\}$;

Function FINDINCORE

Input: $G = (V_G, E_G)$, $\mathcal{K}$, v
Output: d and H

1 Let H be an empty graph; Let $Q \leftarrow \emptyset$;
2 **for** $v \in V_G$ **do**
3 $\mathsf{d}[v] \leftarrow 0$; $\mathtt{visited}[v] \leftarrow false$;
4 Let x be an integer and $x \leftarrow k_{max}(v,G)$;
5 Q.enqueue(v); $\mathtt{visited}[v] \leftarrow true$;
6 **while** $Q \neq \emptyset$ **do**
7 $v \leftarrow Q$.dequeue(); $V_H \leftarrow V_H \cup \{v\}$;
8 **for** $(w,v) \in E_G$ **do**
9 **if** $k_{max}(w,G) \geq x$ **then**
10 $\mathsf{d}[v] \leftarrow \mathsf{d}[v] + 1$;
11 **if** $k_{max}(w,G) = x$ ***and*** $\mathtt{visited}[w]$ *is* $false$ **then**
12 Q.enqueue(w); $E_H \leftarrow E_H \cup \{(w,v)\}$;
13 $\mathtt{visited}[w] \leftarrow true$;

Given $M = min\{k_{max}(u_1,G'), k_{max}(v_1,G')\}$, the algorithm first computes the $(i,0)$-cores for both G and $G' = G \oplus (u_1,v_1)$, where $0 \leq i \leq M$ (lines 2-4). Note that vertices v in these subgraphs may have $l_{max}(v,k,G)$ changed according to Theorem 4.6, based on which Algorithm 3 maintains $l_{max}(v,i,G)$ with $0 \leq i \leq M-1$ by inserting (u_1,v_1) into G_i (lines 5-6). For the $(M,0)$-core, k_{max} of vertices may be incremented if $k_{max}(u_1,G) \geq k_{max}(v_1,G)$, leading to the insertion of some edges $\mathcal{E}$ besides (u_1,v_1) into $(M,0)$-core. The algorithm identifies these edges first and maintains $l_{max}(v,M,G)$

for $v \in V_{G'_M}$ by inserting them into G_M (lines 7-11). Note that the maintenance of $l_{max}(k)$ for different k values ($0 \le k \le M-1$) is independent of each other, providing an opportunity for parallel execution. Specifically, for lines 5-6 of Algorithm 3, we can allocate a thread to each k for the parallel maintenance of the corresponding $l_{max}(k)$. Since the detailed maintenance of $l_{max}(k')$ with given k' is a trivial extension of the k_{max} maintenance by considering only vertices in $(k', 0)$-core, we have omitted the details for brevity.

Complexity Analysis. We analyze the complexity of Algorithms 1, 2, and 3. Algorithm 2 first calculates $ED(v, G)$ and $PED(v, G)$ for each vertex v, which takes $O(|E_G|)$ time. Then, it performs a DFS search on vertices v with $\mathsf{d}[v]$ larger than x and removes vertices v if $\mathsf{d}[v] \le x$, which transverses the whole graph in the worst case. Hence, Algorithm 2 has a time complexity of $O(|E_G|)$. Since M is bounded by $E_G^{0.5}$ [13], and $l_{max}(k)$ maintenance for a specific k costs $O(|E_G|)$ time, Algorithm 3 has a time complexity of $O(|E_G^{1.5}|)$. Putting them together, Algorithm 1 takes $O(|E_G^{1.5}|)$ time as well. The space complexities of Algorithms 1, 2, and 3 are $O(|E_G|)$. Since M is small in real-world networks, as will be verified in experiments, Algorithm 1 has a time complexity close to $O(|E_G|)$ and is practically efficient.

5.2 Local-search-based D-core Maintenance Algorithm for a Single-edge Deletion

The algorithm for handling a single-edge deletion also maintains $k_{max}(v, G)$ and $l_{max}(v, k, G)$ in a two-stage fashion as Algorithm 1. The maintenance of $k_{max}(v)$ for each v under a single-edge deletion is reported in Algorithm 4. The algorithm first does the initialization (lines 1-5) and then collects candidates v that may have their $k_{max}(v)$ decremented (i.e., vertices in in-core group of r based on Theorems 4.2 and 4.3) using BFS (lines 6-11). If $k_{max}(u_1) < k_{max}(v_1)$, Algorithm 4 stops the maintenance based on Theorem 4.2 (line 4-5). Otherwise, a decomposition-like procedure is performed to update k_{max} for the collected vertices (lines 12-21) [6]. The deletion maintenance for $l_{max}(k)$ follows a structure similar to Algorithm 3, and we omit the details for the sake of concision.

Example 5.2. *Reconsider the digraph G in Figure 1, but assume that (v_2, v_8) is the edge to be deleted. We illustrate Algorithm 4. The algorithm first chooses v_8 as r. Since $k_{max}(v_2) = k_{max}(v_8)$, the algorithm calculates H_1 , H_2, d_1, and d_2 using Function FINDINCORE. Then, the subgraph H composed by $\{v_2, v_3, v_5, v_6, v_7, v_8\}$ is obtained. Since all vertices $v \in V_H$ have $\mathsf{d}[v]$ larger than 3, no vertices will have $k_{max}(v)$ decremented.*

Complexity Analysis. We analyze the complexity of Algorithm 4. It first finds the in-core group of r by BFS and calculates $\mathsf{d}[v]$ of each vertex v in it. This takes $O(|E_G|)$ time. Then, this algorithm removes vertices v without enough $\mathsf{d}[v]$ in H iteratively, which also takes $O(|E_G|)$ time. Therefore, Algorithm 4 has a time complexity of $O(|E_G|)$. The space complexity is $O(|E_G|)$.

5.3 Optimizations

We further propose three optimizations to improve the performance of maintaining $l_{max}(k)$. Note that all optimizations can be similarly applied to both the edge insertion and deletion cases. For brevity, we only present the details for the edge insertion case.

Optimization 1: Pruning identical $(k, 0)$-cores.
As indicated by Algorithm 3, with obtained $\{k_{max}(v, G') : \forall v \in V_{G'}\}$, it inserts (u_1, v_1) into $(k, 0)$-cores ($0 \le k \le M-1$) to maintain $l_{max}(v, k, G)$. However, we observe that a lot of $(k, 0)$-cores with different k values have identical structures. In specific,

Theorem 5.1. *Consider a digraph G, $k_{max}(v)$ of all vertices $v \in V_G$ constitute an integer set $\mathcal{I}$ where each item can be sorted in ascending order. Specifically, $\mathcal{I} = \{I_1, I_2, \cdots\}$, where each I_i is unique, $I_i < I_{i+1}$, and $|\mathcal{I}| \le |V_G|$. Then, for two adjacent integers I_l and I_u with $I_l - I_u > 1$, the $(k, 0)$-cores with $I_l < k \le I_u$ have identical structures. If $I_1 \ne 0$, then $(k, 0)$-cores with $0 \le k \le I_1$ have identical structures.*

Take digraph G in Figure 1 as an example. The integer set $\mathcal{I}$ is $\{2, 3\}$, as $\forall v \in V_G$, either $k_{max}(v) = 2$ or $k_{max}(v) = 3$. Hence, $(0, 0)$-core, $(1, 0)$-core, and $(2, 0)$-core have identical structures. For $(k, 0)$-cores with identical structures but different k values, once we complete the $l_{max}(k)$ maintenance for any one of the $(k, 0)$-cores, we can skip the processing for the remaining ones. For example, once we complete the processing for the $(2, 0)$-core and obtain $l_{max}(v, 2, G')$ for each vertex v in G' of Figure 1, we can set $l_{max}(v, 0) = l_{max}(v, 1) = l_{max}(v, 2)$ and skip the maintenance of $l_{max}(v, k)$ for the $(0, 0)$-core and $(1, 0)$-core. We apply Optimization 1 in lines 5-6 of Algorithm 3 to improve the efficiency.

Optimization 2: Skipping redundant $l_{max}(v, k)$ maintenance. The rationale behind Optimization 2 is that for a vertex v with identical $l_{max}(v, k)$ values for a set of consistent $k \in [k_{low}, k_{high}]$, if v has $l_{max}(v, k_2)$ incremented, then it will definitely have $l_{max}(v, k_1)$ incremented as well. Here, $k_{low} \le k_1 \le k_2 \le k_{high}$. The reason is that $(k_2, 0)$-core $\subseteq (k_1, 0)$-core based on Property 3.1. Specifically,

Theorem 5.2. *Given a digraph G, an inserted edge (u_1, v_1), and a vertex v having a set of identical $l_{max}(v, k, G)$ with the consistent k, where $k \in [k_{low}, k_{high}]$. If v has one of its $l_{max}(v, k_2)$ incremented by 1, where $k_2 \in [k_{low}, k_{high}]$, we can directly increment all the $l_{max}(v, k_1)$ with $k_{low} \le k_1 \le k_2$ by 1.*

By maintaining $l_{max}(v, k)$ for each $v \in V_{G'}$ in the *descending* order of k in lines 5-6 of Algorithm 3, we can fully utilize Optimization 2. For example, given a vertex v and $\Phi(v) = \{(0, 3), (1, 2), (2, 2), (3, 2)\}$, we can see that for $k \in [1, 3]$, $l_{max}(v, k)$ are the same. Assume $l_{max}(v, 3)$ is incremented to 3 with a single-edge insertion. We can directly increase $l_{max}(v, 1)$ and $l_{max}(v, 2)$ by one as well according to Theorem 5.2. Note that this observation may not hold for the edge deletion case. Hence, we do not use this optimization when dealing with edge deletions.

Optimization 3: Reusing maintenance results.
In lines 5-6 of Algorithm 3, we process each different $(k, 0)$-core from scratch. However, based on Property 3.1, the $(k, 0)$-cores with larger k are essentially subgraphs of those with smaller k. Based on this observation, when maintaining $l_{max}(k)$ with different k values in descending order, the results generated during the maintenance of $(k, 0)$-cores with larger k can be reused. For example, for the initialization of the adjacent lists for each $(k, 0)$-core in lines 5-6 of Algorithm 3, we can use the adjacent lists from previous $(k_2, 0)$-cores, and compute the new adjacent lists and based on new vertices V_N in current $(k_1, 0)$-cores, i.e., $V_N = V_{(k_1,0)-\text{core}} \setminus V_{(k_2,0)-\text{core}}$. Here, $k_2 > k_1$.

6 ALGORITHMS FOR BATCH UPDATES

In this section, we first present the edge grouping strategies facilitating D-core maintenance with batch updates. We then introduce an H-index-based k_{max} maintenance algorithm for batch insertions. Finally, we present the overall H-index-based D-core maintenance algorithm for edge insertions and its extension for edge deletions.

Motivation. In real-world applications, scenarios involving simultaneous insertions or deletions of multiple edges are prevalent. A straightforward approach to handling D-core maintenance in such situations is to process these edges one by one using the proposed local-search-based algorithms. However, this method is inefficient for handling batch updates, as it unnecessarily revisits vertices, resulting in a linear increase in running time as the number of updated edges grows. Even worse, the affected vertices are handled sequentially for each updated edge, which further worsens the performance. To address these concerns, we propose an H-index-based algorithm that processes multiple affected vertices simultaneously, thereby enhancing overall efficiency. Additionally, we introduce two edge grouping strategies to facilitate the simultaneous processing of multiple edges. These strategies identify groups of edges for which the insertion or deletion will cause a $k_{max}(v, G)$ ($l_{max}(v, k, G)$) change of at most 1 for the affected vertex v. By identifying such edge groups and inserting/deleting them at the same time, the efficiency can be further enhanced by precisely initializing the vertex values for affected vertices close to the updated ones.

6.1 The Edge Grouping Strategies

We start by presenting the edge-grouping strategies. Note that the detailed algorithm for constructing edge groups is omitted in this paper due to space limitations and can be found in the technical report. Those strategies are applicable to both k_{max} and $l_{max}(k)$ similarly, and we only present the details based on k_{max} for brevity. To simplify the presentation, we denote the smaller k_{max} of the two endpoints of an edge as the k_{max} value of that edge, i.e., $k_{max}((u, v)) = min\{k_{max}(u), k_{max}(v)\}$.

Strategy 1: Group by $k_{max}((u, v))$. As Theorem 4.1 suggests, the insertion or deletion of an edge can lead to a change in $k_{max}(v)$ of any vertex v by at most 1. Consequently, edges with different k_{max} can be grouped together, allowing for simultaneous insertions or deletions of edges in the same group. This principle is detailed in the following theorem.

Theorem 6.1. *Consider a digraph G and a collection of edges E_i to be inserted/deleted. To improve the efficiency, we categorize these edges into distinct groups denoted as $\mathcal{B} = \{B_1, B_2, \ldots, B_j\}$. For any two edges e_1 and e_2 within B_i $(1 \le i \le j)$, $|k_{max}(e_1) - k_{max}(e_2)| > 1$ holds. Then, the insertion/deletion of B_i leads to a change in $k_{max}(v)$ of any vertex $v \in V_G$ by at most 1.*

Theorem 6.1 allows edges with different k_{max} values to be grouped and processed together, which helps improve efficiency. However, when dealing with multiple updated edges simultaneously, a common scenario is that these edges follow a power-law distribution w.r.t. their k_{max} values. This implies that numerous edges have the same k_{max}, consequently leading to the creation of many groups containing very few edges, or in some cases, only a single edge, which severely limits the capability for parallel processing.

Strategy 2: Group based on homogeneous edge group. We propose Strategy 2 to handle large sets of edges with the same k_{max}. Recall that the exceptional degree $ED(v, G)$ of a vertex v indicates v's number of in-neighbors in $(k_{max}(v), 0)$-core of G. If we can identify a group of edges for which their insertion would cause the $ED(v, G)$ to increase by at most 1, then the insertion of these edges will also result in $k_{max}(v, G)$ increasing by at most one. To this end, we define the homogeneous edge group as follows.

Table 1: Update of exceptional degree $ED(v, G)$ and pure exceptional degree $PED(v, G)$ for vertices in Figure 2(a)

		Vertex							
		v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
G	$ED(v)$	2	2	2	2	1	1	0	1
	$PED(v)$	0	0	0	0	0	0	0	1
$G' = G \oplus \{(v_3, v_5), (v_4, v_8)\}$	$ED(v)$	2	2	2	2	1	1	0	1
	$PED(v)$	0	0	0	0	0	0	0	1
$G' \oplus \{(v_5, v_7)\}$	$ED(v)$	2	2	2	2	**2**	1	**1**	**2**
	$PED(v)$	0	0	0	0	**1**	**1**	**1**	**0**

Definition 6.1. ***(Homogeneous edge group).*** *A homogeneous edge group $\mathcal{E}_k$ is a batch of edges such that:*

- *$\forall e_i \in \mathcal{E}_k$, $k_{max}(e_i) = k$.*
- *if $\exists e_1, e_2 \in \mathcal{E}_k$ and $e_1 \cap e_2 = w$, then $k_{max}(w, G) > k$.*

Let $V_{\mathcal{E}_k}$ be the vertex set consisting of endpoints of edges $e \in \mathcal{E}_k$. When $\mathcal{E}_k$ is inserted/deleted from a digraph G, $\forall v \in V_{\mathcal{E}_k}$, at most one in-neighbor w of v with $k_{max}(w, G) > k_{max}(v, G)$ is inserted or deleted. We then present the theorem demonstrating how the insertion/deletion of a homogeneous edge group affects $\{k_{max}(v) : \forall v \in V_G\}$.

Theorem 6.2. *Consider a digraph G and a homogeneous edge group $\mathcal{E}_k$ to be inserted/deleted, then $\forall v \in V_G$, only v with $k_{max}(v) = k$ may have $k_{max}(v)$ changed, and this change is no more than 1.*

Example 6.1. *We use Figure 2(a) to illustrate the edge-grouping strategies. The number along a vertex v is $k_{max}(v, G)$. It can be calculate that $k_{max}((v_4, v_8)) = 1$, $k_{max}((v_3, v_5)) = 1$, and $k_{max}((v_5, v_7)) = 0$. Since $|1-0| = 1$ is no greater than 1, no edge groups will be generated based on Strategy 1. However, since $k_{max}((v_4, v_8)) = k_{max}((v_3, v_5)) = 1$ and they do not share any endpoint, they can be accommodated in the same edge group by Strategy 2. Hence, two edge groups are generated based on the three inserted edges, i.e., $B_1 = \{(v_3, v_5), (v_4, v_8)\}$ and $B_2 = \{(v_5, v_7)\}$.*

6.2 The H-index-based $k_{max}(v)$ Maintenance Algorithm for a Batch Insertion

We then propose an H-index-based algorithm for k_{max} maintenance with a batch insertion. As the algorithm for $l_{max}(k')$ with specific k' is similar, it is omitted for brevity. We begin with the definition of H-index [20]. Specifically, the H-index of a collection of integers S is the maximum integer h such that S has at least h integer elements whose values are no less than h, denoted as $\mathcal{H}(S)$. For example, given $S = \{1, 2, 3, 3, 4, 6\}$, H-index $\mathcal{H}(S) = 3$, as S has at least three elements whose values are no less than 3. Based on H-index, we present the definition of *n-order in-H-index* for the digraph.

Definition 6.2. ***(n-order in-H-index [29]).*** *Given a vertex v in G, the n-order in-H-index of v, denoted by $\mathrm{iH}_G^{(n)}(v)$, is defined as*

$$\mathrm{iH}_G^{(n)}(v) = \begin{cases} deg_G^{in}(v), & n = 0 \\ \mathcal{H}(I), & n > 0 \end{cases} \tag{1}$$

where the integer set $I = \{\mathrm{iH}_G^{(n-1)}(u) | u \in N_G^{in}(v)\}$.

Table 2: An illustration of D-core maintenance using Algorithm 5 on graph G in Figure 2.

		Vertex							
		v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
Inserting $\{(v_3, v_5), (v_4, v_8)\}$	original $k_{max}(v)$	2	2	2	2	1	1	0	1
	$\text{iH}^{(0)}(v)$	2	2	2	2	1	1	0	**2**
	$\text{iH}^{(1)}(v) = k_{max}(v)$	2	2	2	2	1	1	0	2
Inserting $\{(v_5, v_7)\}$	$k_{max}(v)$ after inserting $\{(v_3, v_5), (v_4, v_8)\}$	2	2	2	2	1	1	0	2
	$\text{iH}^{(0)}(v)$	2	2	2	2	1	1	**1**	2
	$\text{iH}^{(1)}(v) = k_{max}(v)$	2	2	2	2	1	1	1	2

Theorem 6.3 (Convergence [29]).

$$k_{max}(v, G) = \lim_{n \to \infty} \text{iH}_G^{(n)}(v) \tag{2}$$

Theorem 6.3 shows that $\text{iH}_G^{(n)}(v)$ finally converges to $k_{max}(v, G)$. Based on n-order in-H-index, a natural idea for D-core maintenance with batch updates is initializing $\text{iH}_{G'}^{(0)}(v)$ to $deg_{G'}^{in}(v)$ for each vertex $v \in V_{G'}$ and computing $k_{max}(v, G')$ iteratively. However, this is essentially computing $k_{max}(v, G')$ from scratch, leading to inefficiency for D-core maintenance. Hence, we present how to perform the vertex value initialization based on the theorems in Section 4 and the strategies in Section 6.1 to enhance the efficiency.

Vertex value initialization for a batch insertion. Based on Theorems 4.2, 4.3, and 4.5, given an inserted edge (u_1, v_1), if $k_{max}(u_1, G) \geq k_{max}(v_1, G)$, only vertices v in $C^{in}(v_1)$ with $PED(v, G') > k_{max}(v, G)$ may have $k_{max}(v, G)$ incremented, and the change is at most one based on Theorem 4.1. In addition, Theorems 6.1 and 6.2 show that we can find a group of edges and insert them simultaneously while making sure that $k_{max}(v)$ of each $v \in V_G$ to change no more than one. Based on these theorems, we initialize the vertex values for k_{max} maintenance with an inserted edge group B_i constructed based on strategies in Section 6.1 as follows:

1) For vertex $v \in \bigcup_{\{(u_1,v_1)\} \in B_i} C^{in}(v_1)$, if $\{PED(v, G') > k_{max}(v, G)\} \wedge \{k_{max}(u_1, G) \geq k_{max}(v_1, G)\}$, initialize v's vertex value as $k_{max}(v, G) + 1$, and mark it as active.
2) For vertices v in G that are not marked as active, initialize their vertex values as $k_{max}(v, G)$.

We only mark vertex v as active if v may have $k_{max}(v)$ updated and perform computation over it. The details for the H-index-based algorithm maintaining k_{max} with a batch insertion are outlined in Algorithm 5. The algorithm first does the initialization for each vertex v (lines 1-4). Then, it calculates r (lines 5-8), assigns the initial vertex value, and marks vertices as active when necessary based on our vertex value initiation strategy (lines 9-13). Next, for vertex v needs computation, the algorithm updates the n-order in-H-index of $v' \in N_{G'}^{in}(v)$ (line 20). If $\mathcal{H}(I) < \text{iH}(v)$, Algorithm 5 updates the n-order in-H-index of v and set $flag$ to $true$ to start the next round of computation (lines 21-22). The algorithm finishes the computation and returns $\text{iH}(v)$ as $k_{max}(v, G')$ for each $v \in V_{G'}$ when no vertex w has $\text{iH}(w)$ changed.

Example 6.2. *We use digraph G in Figure 2 to illustrate Algorithm 5, whose calculation process is shown in Table 2. Table 1 presents the update of $ED(v, G)$ and $PED(v, G)$ for each vertex v in Figure 2(a). Take v_8 as an example. First, we insert $\{(v_3, v_5), (v_4, v_8)\}$ and initialize v_8's vertex value as $k_{max}(v_8)+1$, i.e., $\text{iH}^{(0)}(v_8) = 2$, as $PED(v_8) = 2 > k_{max}(v_8) = 1$. Then, Algorithm 5 iteratively computes $\text{iH}^{(n)}(v_8)$. After one iteration, the 1-order in-H-index of v_8 has converged to $\mathcal{H}(\text{iH}^{(0)}(v_3), \text{iH}^{(0)}(v_4)) = \mathcal{H}(2, 2) = 2$. Thus, $k_{max}(v_8)$ is updated as 2. Similarly, $k_{max}(v_7)$ is updated as 1 with the insertion of $\{(v_5, v_7)\}$.*

Algorithm 5: H-index-based $k_{max}(v, G)$ maintenance for each vertex $v \in V_{G'}$ with a batch insertion

Input: digraph $G = (V_G, E_G)$, inserted edge group B_i constructed based on strategies in Section 6.1, original $k_{max}(v, G)$ $\mathcal{K} = \{k_{max}(v, G) : v \in V_G\}$
Output: updated $k_{max}(v, G)$ $\mathcal{K}' = \{k_{max}(v, G') : \forall v \in V_{G'}\}$

```
1  Let G' be a graph with V_G' = V_G and E_G' = E_G ∪ B_i;
2  for v ∈ V_G' do
3      active[v] ← false;
4      Calculate the ED(v, G') and PED(v, G') accordingly;
5  for (u_1, v_1) ∈ B_i do
6      Let r be a vertex and r ← v_1;
7      if k_max(u_1, G) < k_max(v_1, G) then
8          continue;
9      for v ∈ V_G' do in parallel
10         if v ∈ C^in(r) ∧ PED(v, G') > k_max(v, G) then
11             iH(v) ← k_max(v, G) + 1; active[v] ← true;
12         else
13             iH(v) ← k_max(v, G); active[v] ← false;
14 flag ← true;
15 while flag do
16     flag ← false;
17     for v ∈ V_G' do in parallel
18         if active[v] then
19             for each v' ∈ N_G'^in(v) do
20                 I[v'] ← iH(v');
21             if H(I) < iH(v) then
22                 flag ← true, iH(v) ← H(I);
23 return K' = {k_max(v, G') ← iH(v) : ∀v ∈ V_G'};
```

6.3 The Overall H-index-based Algorithm

Similar to Algorithm 1, the complete H-index-based algorithm for a batch insertion also maintains k_{max} and $l_{max}(k)$ sequentially. The details are presented in Algorithm 6, which consists of a k_{max} maintenance step (lines 1-11) and a $l_{max}(k)$ update step (lines 12-21). First, the algorithm constructs the edge groups $\mathcal{B}$ for maintenance of $\mathcal{K} = \{k_{max}(v) : \forall v \in V_G\}$ based on Strategy 1 and 2 (line 1). Next, for each generated group B, the algorithm maintains $\mathcal{K}$ using Algorithm 5, records the maximum k_{max} of edges in $\mathcal{B}$ with M_{max} , and inserts it into G (lines 3-7). The maximum k_{max} of edges in E_i will be recorded then (line 10), and edges in E_i that cannot be processed in parallel are handled sequentially (lines 11). After the maintenance of k_{max}, the algorithm proceeds to maintain $l_{max}(k)$ with $0 \leq k \leq M_{max} + 1$ in a similar manner. Initially, it calculates the difference between the original $(k, 0)$-core G_k and the updated $(k, 0)$-core G'_k as E_D (lines 12-14). Then, it constructs the edge groups $\mathcal{A}$ for parallel $l_{max}(k)$ maintenance, based on E_D and $l_{max}(k)$ of the vertices (line 15). For each edge group A, the al-

Algorithm 6: H-index-based D-core maintenance algorithm with a batch insertion

Input: digraph $G = (V_G, E_G)$, inserted edges E_i, original anchored corenesses $\Phi(G)$
Output: updated anchored corenesses $\Phi(G')$ for G' with $V_{G'} = V_G$ and $E_{G'} = E_G \cup E_i$

1 Construct edge groups $\mathcal{B}$ based on strategies in Section 6.1 with E_i and $\mathcal{K}$, where $\mathcal{K} = \{k_{max}(v, G) : \forall v \in V_G\}$;
2 $M_{max} \leftarrow 0$;
3 **for** $B \in \mathcal{B}$ **do**
4 $\mathcal{K} \leftarrow$ Algorithm 5(G, B, $\mathcal{K}$);
5 **for** *each edge* $(u_1, v_1) \in B$ **do**
6 $M_{max} \leftarrow$ $max\{M_{max}, min\{k_{max}(u_1, G'), k_{max}(v_1, G')\}\}$;
7 Insert B into G;
8 **for** *each edge* $(u_1, v_1) \in E_i$ *but* $\notin \mathcal{B}$ **do**
9 Insert (u_1, v_1) into G;
10 $M_{max} \leftarrow$ $max\{M_{max}, min\{k_{max}(u_1, G'), k_{max}(v_1, G')\}\}$;
11 $\mathcal{K}' \leftarrow$ Algorithm 2(G, (u_1, v_1), $\Phi(G)$);
12 **for** $k \leftarrow M_{max} + 1$ ***to*** 0 **do**
13 Let G_k and G'_k be the $(k, 0)$-core of G and $G \oplus E_i$;
14 Get the difference between G_k and updated G'_k as E_D;
15 Construct edge groups $\mathcal{A}$ with E_D and $l_{max}(k)$ of vertices based on strategies similar to those in Section 6.1;
16 **for** $A \in \mathcal{A}$ **do**
17 Insert/delete A into/from G_k;
18 Maintain $l_{max}(v, k)$ for each vertex $v \in V_{G'_k}$ with a $l_{max}(v, k)$ maintenance algorithm similar to the $k_{max}(v)$ maintenance algortihm in Algorithm 5;
19 **for** *edge* $e' \in E_D$ *but* $\notin \mathcal{A}$ **do**
20 Insert/delete e' into/from G_k;
21 Maintain $l_{max}(v, k)$ for each vertex $v \in V_{G'_k}$ similarly to Algorithm 2;
22 **return** the updated anchored corenesses as $\Phi(G')$;

gorithm inserts it into or deletes it from G_k and maintains $l_{max}(k)$ using an algorithm similar to Algorithm 5 (lines 16-18). Sequential processing is performed for edges in E_D that cannot be processed in parallel (lines 19-21). Note that since the size relationship between G_k' and G_k can not be determined, both edge insertions and deletions are possible in lines 17 and 20.

Complexity analysis. Let the maximum in-degree $\Delta_{in} = \max_{v \in V_G} deg^{in}_G(v)$, and R be the number of convergence rounds required by Algorithm 5. We analyze the complexity of Algorithms 5 and 6. The edge construction based on two strategies can be done in $O(|E_i|)$ time. Algorithm 5 takes $O(R \cdot |V_G| \cdot \Delta_{in} + E_G)$ time in the worst case. Based on these, Algorithm 6 has a time complexity of $O(M_{max} \cdot |E_i| \cdot (R \cdot |V_G| \cdot \Delta_{in} + |E_G|))$. The reason is that the maintenance of k_{max} is bounded by $O(|E_i| \cdot (R \cdot |V_G| \cdot \Delta_{in} + |E_G|))$, and the maintenance cost of each $(k, 0)$-core, i.e., lines 13-21 of Algorithm 6, is definitely smaller than the cost of k_{max} maintenance.

Since our edge grouping strategies can efficiently identify edges for parallel processing, as will be verified in experiments, and we do not need to compute ED and PED values of vertices from scratch every time, the time complexity of Algorithm 6 is close to $O(M_{max} \cdot |\mathcal{B}| \cdot R \cdot |V_G| \cdot \Delta_{in})$.Hence, Algorithm 6 is efficient in practice.

Edge deletion extension of Algorithm 6. We further discuss how to extend the H-index-based algorithm for edge insertions in Algorithm 6 to the edge deletion case. Since the local-search-based algorithm handling edge deletions has been introduced in Section 5.2, our focus is on extending Algorithm 5 to the edge deletion case. This can be easily achieved by modifying the vertex initialization in Section 6.2. Specifically, based on Theorems 4.1, 4.2, 4.3 and 4.4, we should initialize v's vertex value as $k_{max}(v, G) - 1$ for vertex $v \in \bigcup_{\{(u_1,v_1)\} \in B_i} C^{in}(v_1)$, if $\{ED(v, G') > k_{max}(v, G)\} \wedge \{k_{max}(u_1, G) \geq k_{max}(v_1, G)\}$. Here, B_d is the edge set containing edges to be deleted.

7 PERFORMANCE EVALUATION

In this section, we empirically evaluate our proposed algorithms. All experiments are conducted on a Linux server using an Intel Xeon Gold 6330 2.0GHz CPU with a total of 56 cores with two-way hyper-threading and 128 GB of memory, running Oracle Linux 8.6. Our algorithms are implemented in C++, compiled with the g++ compiler at -O3 optimization level, and parallelized using OpenMP.

Datasets. We use nine real-world graphs in our experiments. Table 3 shows the statistics of these graphs. Specifically, `P2p`[3] is a peer-to-peer file sharing graph; `Email-EuAll`[3] is a communication graph; `Amazon`[3] is a product co-purchasing graph; `Stack Overflow`[3] is an internet interaction graph; `Hollywood`[4] is a collaboration graph of actors; `Message`[3], `Slashdot`[3], `Pokec`[3], and `Live Journal`[3] are social graphs.

Algorithms. We compare several algorithms in our experiments.

(1) **DECOMP** [13]: The state-of-the-art D-core decomposition algorithm.
(2) **PEEL** [13]: The state-of-the-art peeling-based D-core maintenance algorithm.
(3) **LOCAL**: Our proposed local-search-based D-core maintenance algorithm with all the proposed optimizations.
(4) **HINDEX**: Our proposed H-index-based D-core maintenance algorithm with all the optimizations and edge-grouping strategies.

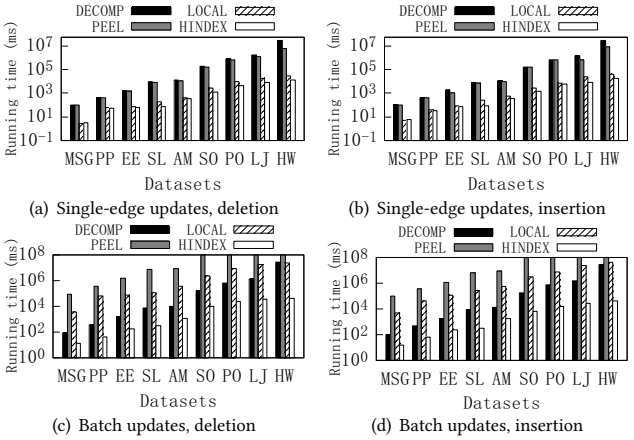
Parameters and metrics. The parameters tested in experiments include the number of threads, the number of updated edges ($|\Delta G|$), and the graph size, with default settings of 16, 1K, and $100\% \cdot |E_G|$, respectively. We randomly generate edges to be inserted/deleted. Two update modes are considered, i.e., single-edge updates and batch updates. For single-edge updates, we generate 1,000 updates in each experiment and report the average results. For batch updates, we randomly insert or delete the default number of edges and report the total processing time. The performance metrics evaluated include the running time (in milliseconds), the improvement brought by optimizations in Section 5.3 (in percentage), and the number of edges grouped by strategies in Section 6.1. We halt the algorithms' execution when the running time exceeds 24 hours.

Exp-1: Our algorithms vs. existing methods. We start by comparing the performance of our proposed algorithms with DECOMP and PEEL under the default experiment settings. Figure 4 reports the results. We can see that our algorithms are much more efficient than the SOTA. For single-edge updates, PEEL costs over 2.5 hours to process the largest dataset `HW` with over 200 million edges, while

[3] http://snap.stanford.edu/data/index.html
[4] http://law.di.unimi.it/datasets.php

Table 3: Statistics of the datasets (deg_{avg} represents the average degree; K = 10^3, and M = 10^6)

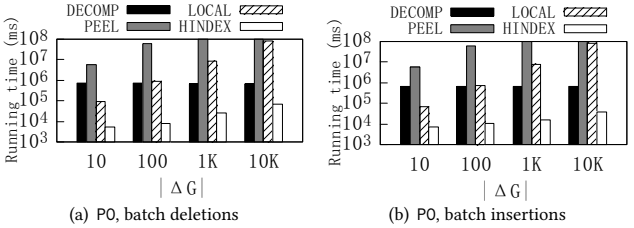
Dataset	Abbr.	$\|V_G\|$	$\|E_G\|$	deg_{avg}	k^G_{max}	l^G_{max}
Message	MSG	1.9**K**	20.3**K**	10.69	14	14
P2p	PP	62.6**K**	147.9**K**	2.36	1	2
Email-EuAll	EE	265.2**K**	420**K**	1.58	27	26
Slashdot	SL	82.1**K**	948.4**K**	11.54	53	53
Amazon	AM	400.7**K**	3.2**M**	7.99	10	10
Stack Overflow	SO	2.5**M**	16.3**M**	6.60	19	17
Pokec	PO	1.6**M**	30.6**M**	18.75	32	31
Live Journal	LJ	4.8**M**	69.0**M**	14.23	252	253
Hollywood	HW	2.1**M**	228.9**M**	105.00	1,297	99



(a) Single-edge updates, deletion (b) Single-edge updates, insertion

(c) Batch updates, deletion (d) Batch updates, insertion

Figure 4: Performance comparisons with the SOTA

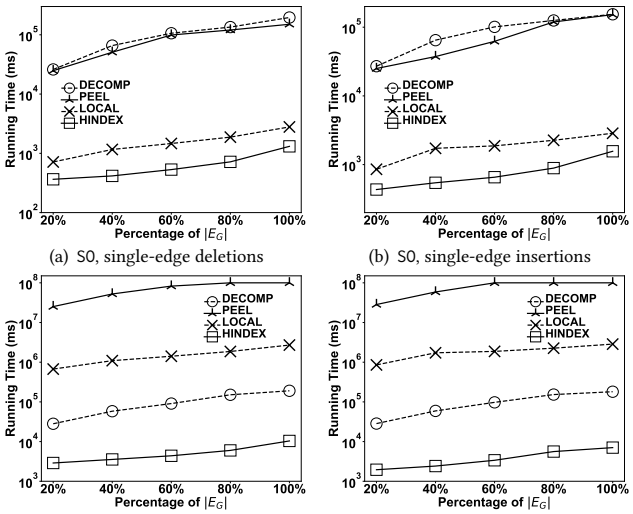


(a) PO, batch deletions (b) PO, batch insertions

Figure 5: Effect of the number of updated edges $|\Delta G|$

our methods take less than 1 minute. Besides, our methods are up to two orders of magnitude faster than PEEL on the other datasets. For batch updates, PEEL cannot finish the processing within 24 hours over the large graphs with more than 10 million edges, including SO, PO, LJ, and HW, while our algorithm HINDEX can still finish within 1 minute. Besides, HINDEX outperforms PEEL by up to 20,000x over the other datasets and is up to 700x faster than DECOMP over all datasets. These well demonstrate the superior performance of our proposed algorithms. Moreover, HINDEX is much more efficient than LOCAL for batch updates. For example, for the large graphs with more than 30 million edges, including PO, LJ, and HW, the improvement is up to three orders of magnitude. The reason is that when there are multi-edge updates, HINDEX can insert/delete multiple edges and efficiently compute the updated corenesses for multiple affected vertices simultaneously. On the other hand, LOCAL has to process every edge one by one, which deteriorates its performance for handling batch updates.

Exp-2: Effect of the number of updated edges on batch updates. Next, we vary the number of updated edges $|\Delta G|$ from 10 to 10K and examine the impact on the performance of the algorithms



(a) SO, single-edge deletions (b) SO, single-edge insertions

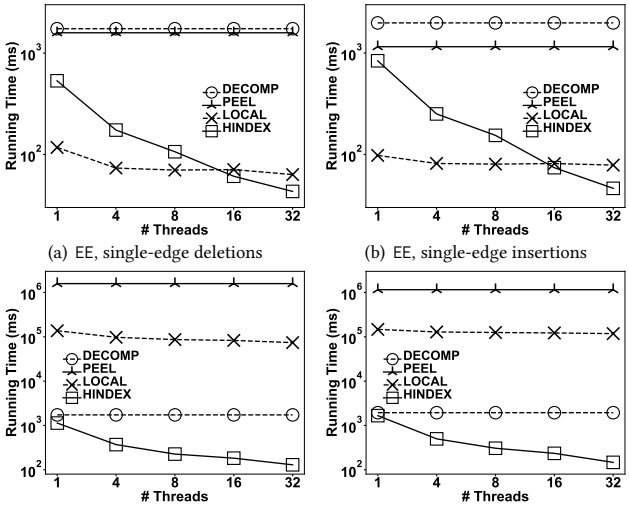
(c) SO, batch deletions (d) SO, batch insertions

Figure 6: Effect of cardinality

for batch updates over the PO dataset. As shown in Figure 5, with the increase of $|\Delta G|$, the running time of DECOMP is almost unchanged, as its cost is $O(|G \oplus \Delta G|^{1.5})$ [13], which remains stable. For PEEL and LOCAL, as they process batches of edges sequentially, their running time grows linearly with $|\Delta G|$. Nevertheless, LOCAL exhibits better performance than PEEL. For example, LOCAL is nearly 2 orders of magnitude faster than PEEL with the $|\Delta G|$ of 100 for batch insertions. In addition, the running time of HINDEX is much more stable with the increase of $|\Delta G|$ compared to LOCAL and PEEL, making HINDEX very efficient for handling batch updates. For instance, HINDEX outperforms DECOMP up to 20x on Figure 5(b) with an update of 10K edges. On average, HINDEX outperforms DECOMP by 63.7x for edge deletions and 51.2x for edge insertions over all the settings of $|\Delta G|$.

Exp-3: Effect of dataset cardinality. We evaluate the effect of cardinality on all the algorithms using the SO dataset. For this purpose, we extract a set of subgraphs from the original graph by randomly selecting different fractions of edges, ranging from 20% to 100%. As shown in Figure 6, the running time of all the algorithms increases with the dataset cardinality. This is expected, as a larger dataset leads to more updates in the coreness of vertices, resulting in decreased performance. For instance, in Figure 6(b), when the percentage of E_G increases from 20% to 100%, the running time of LOCAL and HINDEX grows from 854 ms to 2,858 ms and from 433 ms to 1,562 ms, respectively.

Exp-4: Effect of the number of threads. We assess the effect of the number of threads on performance by varying the number of threads from 1 to 32. Figure 7 presents the results for the EE dataset. It is evident that all our algorithms exhibit reduced running time as the number of threads increases. This improvement can be attributed to the utilization of more threads, which enables the maintenance of more $l_{max}(v,k)$ simultaneously for each vertex v in LOCAL, and allows more vertices to maintain anchored corenesses concurrently within a single iteration in HINDEX, thereby enhancing overall performance. For instance, in Figure 7(d), HINDEX demonstrates an 11.3x speedup when the number of threads increases from 1 to 32 for the batch insertion case. Additionally,



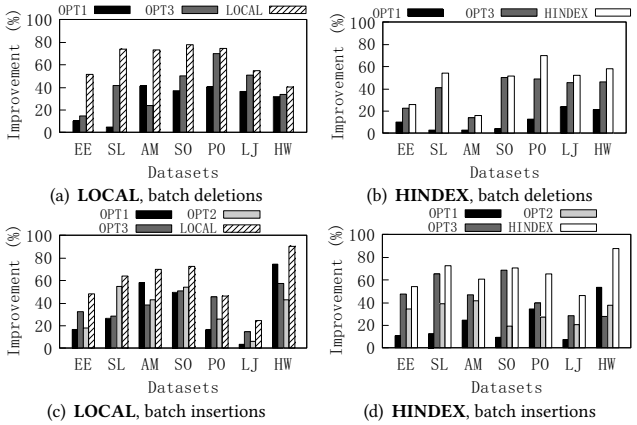
(a) EE, single-edge deletions (b) EE, single-edge insertions

(c) EE, batch deletions (d) EE, batch insertions

Figure 7: Effect of # threads

HINDEX exhibits superior parallelism performance compared to LOCAL. This is because LOCAL parallelizes the computation of $l_{max}(k)$ with different k values, which is relatively small compared to the number of threads, thus limiting the potential performance gains from multiple threads. In contrast, HINDEX parallelizes the computation across different vertices, which takes full advantage of the computing power provided by additional threads. Nevertheless, it is noteworthy that even when using a single thread, HINDEX performs more efficiently than DECOMP for batch updates. Additionally, both our algorithms are faster than PEEL and DECOMP for single-edge updates.

Exp-5: Effect of pruning optimizations. We now show the efficiency of the three optimizations proposed in Section 5.3 for batch updates; similar results are also obtained for single-edge updates. It is worth noting that since the observation made in Optimization 2 does not apply to the edge deletion case, we do not utilize it for edge deletion processing. Besides, the proposed optimizations are also applicable to the H-index-based algorithm. There are five competitor methods, including Optimization 1 (OPT1), Optimization 2 (OPT2), Optimization 3 (OPT3), and the combinations of all of them (LOCAL and HINDEX). Figure 8 reports the efficiency improvement results of all different optimization methods w.r.t. the local-search-based algorithm (the algorithm for edge insertions presented in Algorithm 1) and the H-index-based algorithm (the algorithm for edge insertions presented in Algorithm 6). We observe that all three optimizations are effective and efficient. For instance, in the case of edge insertions, OPT1, OPT2, OPT3, and LOCAL enhance the performance of the local search algorithm by up to 74%, 42%, 57%, and 90% respectively, on the HW dataset. Additionally, OPT1, OPT2, OPT3, and HINDEX improve the H-index-based algorithm by up to 53%, 37%, 28%, and 88% respectively, for edge insertions on the HW dataset. On average, across all reported datasets for edge insertions, OPT1, OPT2, OPT3, and LOCAL enhance the performance of the local algorithm by up to 35%, 35%, 39%, and 60% respectively. Furthermore, for edge insertions, OPT1, OPT2, OPT3, and HINDEX enhance the performance of the H-index-based algorithm by up to 22%, 31%, 47%, and 66% respectively, across all reported datasets.



(a) **LOCAL**, batch deletions (b) **HINDEX**, batch deletions

(c) **LOCAL**, batch insertions (d) **HINDEX**, batch insertions

Figure 8: Effect of pruning optimizations for batch updates

Table 4: Number of edges processable in parallel by proposed edge-grouping strategies

Strategies		Datasets				
		SL	AM	SO	PO	LJ
Strategy 1	$\|\Delta G\|$: 100	13	0	11	6	16
	$\|\Delta G\|$: 1**K**	153	83	114	93	143
	$\|\Delta G\|$: 10**K**	8,230	9,341	1,055	928	1,453
	$\|\Delta G\|$: 100**K**	19,455	4,226	8,494	13,560	18,014
Strategy 2	$\|\Delta G\|$: 100	83	99	89	94	84
	$\|\Delta G\|$: 1**K**	846	916	885	907	857
	$\|\Delta G\|$: 10**K**	1,769	658	8,945	9,072	8,546
	$\|\Delta G\|$: 100**K**	80,545	95,774	91,506	96,440	81,986
Ungrouped	$\|\Delta G\|$: 100	4	1	0	0	0
	$\|\Delta G\|$: 1**K**	1	1	1	0	0
	$\|\Delta G\|$: 10**K**	1	1	0	0	1
	$\|\Delta G\|$: 100**K**	0	0	0	0	0

Exp-6: Effect of edge-grouping strategies. Finally, we evaluate the effect of different edge-grouping strategies for parallel processing in the H-index-based algorithm. Table 4 shows the number of edges processable in parallel based on different strategies. We can observe that our proposed edge-grouping strategies can efficiently identify the edges that can be simultaneously inserted/deleted from the original graph. In addition, we can see that more edges are grouped by Strategy 2 when $|\Delta G|$ is of large size, e.g., 100K. The reason is that in real-world graphs, edges always follow a power-law distribution w.r.t. their k_{max} values, resulting in the majority of updated edges having the same k_{max} value, which leads to the better performance of Strategy 2.

8 CONCLUSION

In this paper, we study the D-core maintenance problem in dynamic digraphs. To address this problem, we introduce novel theoretical findings to identify vertices that require anchored coreness updates. Moreover, we propose a local-search-based algorithm along with three optimizations specifically designed for D-core maintenance following single-edge insertions or deletions. To enhance performance in the case of batch updates, we further propose an H-index-based algorithm incorporating innovative edge-grouping strategies. Theoretical analysis and extensive empirical evaluations demonstrate the effectiveness of our proposed algorithms. Moving forward, our future research will focus on developing efficient *distributed* algorithms to tackle the D-core maintenance problem.

REFERENCES

[1] Babak Abedin and Babak Sohrabi. 2009. Graph theory application and web page ranking for website link structure improvement. *Behaviour & Information Technology* 28, 1 (2009), 63–72.

[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] Sabeur Aridhi, Martin Brugnara, Alberto Montresor, and Yannis Velegrakis. 2016. Distributed k-core decomposition and maintenance in large dynamic graphs. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. 161–168.

[4] 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.

[5] Wen Bai, Yuncheng Jiang, Yong Tang, and Yayang Li. 2022. Parallel Core Maintenance of Dynamic Graphs. *IEEE Transactions on Knowledge and Data Engineering* (2022).

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

[7] Francesco Bonchi, Francesco Gullo, Andreas Kaltenbrunner, and Yana Volkovich. 2014. Core decomposition of uncertain graphs. In *Proceedings of the 20th International Conference on Knowledge Discovery and Data Mining*. 1316–1325.

[8] Yankai Chen, Jie Zhang, Yixiang Fang, Xin Cao, and Irwin King. 2021. Efficient community search over large directed graphs: An augmented index-based approach. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*. 3544–3550.

[9] James Cheng, Yiping Ke, Shumo Chu, and M Tamer Özsu. 2011. Efficient core decomposition in massive networks. In *2011 IEEE 27th International Conference on Data Engineering*. 51–62.

[10] Naga Shailaja Dasari, Ranjan Desh, and Mohammad Zubair. 2014. ParK: An efficient algorithm for k-core decomposition on multicore processors. In *2014 IEEE International Conference on Big Data*. 9–16.

[11] Marius Eidsaa and Eivind Almaas. 2013. S-core network decomposition: A generalization of k-core analysis to weighted networks. *Physical Review E* 88, 6 (2013), 062819.

[12] Hossein Esfandiari, Silvio Lattanzi, and Vahab Mirrokni. 2018. Parallel and streaming algorithms for k-core decomposition. In *International Conference on Machine Learning*. 1397–1406.

[13] Yixiang Fang, Zhongran Wang, Reynold Cheng, Hongzhi Wang, and Jiafeng Hu. 2018. Effective and efficient community search over large directed graphs. *IEEE Transactions on Knowledge and Data Engineering* 31, 11 (2018), 2093–2107.

[14] Yixiang Fang, Yixing Yang, Wenjie Zhang, Xuemin Lin, and Xin Cao. 2020. Effective and Efficient Community Search over Large Heterogeneous Information Networks. *Proceedings of the VLDB Endowment* 13, 6 (2020), 854–867.

[15] Edoardo Galimberti, Martino Ciaperoni, Alain Barrat, Francesco Bonchi, Ciro Cattuto, and Francesco Gullo. 2021. Span-core Decomposition for Temporal Networks: Algorithms and Applications. *ACM Transactions on Knowledge Discovery from Data* 15, 1 (2021), 2:1–2:44.

[16] Sen Gao, Hongchao Qin, Rong-Hua Li, and Bingsheng He. 2023. Parallel Colorful h-Star Core Maintenance in Dynamic Graphs. *Proceedings of the VLDB Endowment* 16, 10 (2023), 2538–2550.

[17] Christos Giatsidis, Dimitrios M Thilikos, and Michalis Vazirgiannis. 2013. D-cores: measuring collaboration of directed graphs based on degeneracy. *Knowledge and information systems* 35, 2 (2013), 311–343.

[18] Bin Guo and Emil Sekerinski. 2022. Simplified Algorithms for Order-Based Core Maintenance. *arXiv preprint arXiv:2201.07103* (2022).

[19] Bin Guo and Emil Sekerinski. 2023. Parallel Order-Based Core Maintenance in Dynamic Graphs. In *Proceedings of the 52nd International Conference on Parallel Processing*. 122–131.

[20] Jorge E. Hirsch. 2005. *H-index*. https://en.wikipedia.org/wiki/H-index

[21] Qiang-Sheng Hua, Yuliang Shi, Dongxiao Yu, Hai Jin, Jiguo Yu, Zhipen Cai, Xiuzhen Cheng, and Hanhua Chen. 2019. Faster parallel core maintenance algorithms in dynamic graphs. *IEEE Transactions on Parallel and Distributed Systems* 31, 6 (2019), 1287–1300.

[22] Qiang-Sheng Hua, Xiaohui Zhang, Hai Jin, and Hong Huang. 2023. Revisiting core maintenance for dynamic hypergraphs. *IEEE Transactions on Parallel and Distributed Systems* 34, 3 (2023), 981–994.

[23] Paul Irofti, Andrei Patrascu, and Andra Baltoiu. 2019. Quick survey of graph-based fraud detection methods. *arXiv preprint arXiv:1910.11299* (2019).

[24] Hai Jin, Na Wang, Dongxiao Yu, Qiang-Sheng Hua, Xuanhua Shi, and Xia Xie. 2018. Core maintenance in dynamic graphs: A parallel approach based on matching. *IEEE Transactions on Parallel and Distributed Systems* 29, 11 (2018), 2416–2428.

[25] Wissam Khaouid, Marina Barsky, Venkatesh Srinivasan, and Alex Thomo. 2015. K-core decomposition of large networks on a single pc. *Proceedings of the VLDB Endowment* 9, 1 (2015), 13–23.

[26] Jon M Kleinberg, Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew S Tomkins. 1999. The web as a graph: Measurements, models, and methods. In *Computing and Combinatorics: 5th Annual International Conference, COCOON'99 Tokyo, Japan, July 26–28, 1999 Proceedings 5*. Springer, 1–17.

[27] 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.

[28] Zhongmou Li, Hui Xiong, Yanchi Liu, and Aoying Zhou. 2010. Detecting blackhole and volcano patterns in directed networks. In *2010 IEEE International Conference on Data Mining*. IEEE, 294–303.

[29] Xuankun Liao, Qing Liu, Jiaxin Jiang, Xin Huang, Jianliang Xu, and Byron Choi. 2022. Distributed D-core decomposition over large directed graphs. *Proceedings of the VLDB Endowment* 15, 8 (2022), 1546–1558.

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

[31] Boge Liu, Long Yuan, Xuemin Lin, Lu Qin, Wenjie Zhang, and Jingren Zhou. 2019. Efficient (α,β)-core Computation: an Index-based Approach. In *International World Wide Web Conference*. 1130–1141.

[32] Qing Liu, Minjun Zhao, Xin Huang, Jianliang Xu, and Yunjun Gao. 2020. Truss-based community search over large directed graphs. In *Proceedings of the 2020 ACM International Conference on Management of Data*. 2183–2197.

[33] Qing Liu, Xuliang Zhu, Xin Huang, and Jianliang Xu. 2021. Local algorithms for distance-generalized core decomposition over large dynamic graphs. *Proceedings of the VLDB Endowment* 14, 9 (2021), 1531–1543.

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

[35] Qi Luo, Dongxiao Yu, Zhipeng Cai, Xuemin Lin, Guanghui Wang, and Xiuzhen Cheng. 2023. Toward maintenance of hypercores in large-scale dynamic hypergraphs. *The VLDB Journal* 32, 3 (2023), 647–664.

[36] Qi Luo, Dongxiao Yu, Zhipeng Cai, Yanwei Zheng, Xiuzhen Cheng, and Xuemin Lin. 2023. Core maintenance for hypergraph streams. *World Wide Web* 26, 5 (2023), 3709–3733.

[37] 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.

[38] Chenhao Ma, Yixiang Fang, Reynold Cheng, Laks VS Lakshmanan, Wenjie Zhang, and Xuemin Lin. 2021. On directed densest subgraph discovery. *ACM Transactions on Database Systems (TODS)* 46, 4 (2021), 1–45.

[39] Fragkiskos D Malliaros, Christos Giatsidis, Apostolos N Papadopoulos, and Michalis Vazirgiannis. 2020. The core decomposition of networks: Theory, algorithms and applications. *The VLDB Journal* 29 (2020), 61–92.

[40] Aritra Mandal and Mohammad Al Hasan. 2017. A distributed k-core decomposition algorithm on spark. In *IEEE International Conference on Big Data*. 976–981.

[41] Alberto Montresor, Francesco De Pellegrini, and Daniele Miorandi. 2012. Distributed k-core decomposition. *IEEE Transactions on Parallel and Distributed Systems* 24, 2 (2012), 288–300.

[42] Georgios A Pavlopoulos, Maria Secrier, Charalampos N Moschopoulos, Theodoros G Soldatos, Sophia Kossida, Jan Aerts, Reinhard Schneider, and Pantelis G Bagos. 2011. Using graph theory to analyze biological networks. *BioData mining* 4 (2011), 1–27.

[43] You Peng, Ying Zhang, Wenjie Zhang, Xuemin Lin, and Lu Qin. 2018. Efficient Probabilistic K-Core Computation on Uncertain Graphs. In *IEEE International Conference on Data Engineering*. 1192–1203.

[44] Derek J De Solla Price. 1965. Networks of scientific papers: The pattern of bibliographic references indicates the nature of the scientific research front. *Science* 149, 3683 (1965), 510–515.

[45] 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.

[46] 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.

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

[48] Anxin Tian, Alexander Zhou, Yue Wang, and Lei Chen. 2022. Maximal D-truss Search in Dynamic Directed Graphs. *Proceedings of the VLDB Endowment* 16, 9 (2022), 2199–2211.

[49] Na Wang, Dongxiao Yu, Hai Jin, Chen Qian, Xia Xie, and Qiang-Sheng Hua. 2017. Parallel algorithm for core maintenance in dynamic graphs. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2366–2371.

[50] Dong Wen, Lu Qin, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2016. I/O efficient Core Graph Decomposition at web scale. In *International Conference on Data Engineering*. 133–144.

[51] Huanhuan Wu, James Cheng, Yi Lu, Yiping Ke, Yuzhen Huang, Da Yan, and Hejun Wu. 2015. Core decomposition in large temporal graphs. In *IEEE International*

Conference on Big Data. 649–658.
[52] 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 THEOREMS

We provide the detailed proofs of theorems in Sections 4, 5, and 6 in the appendix.

Proof of Theorem 4.1. We first prove the insertion case. Assume that after the insertion of (u_1, v_1), $k_{max}(v) = k$ is increased by n to $k+n$ for a vertex v, where $n > 1$. It is obvious that $(u_1, v_1) \in (k+n, 0)$-core, as otherwise the original $k_{max}(v)$ of v is $k + n$, which is a contradiction. Let $Z = (k + n, 0)$-core$\backslash(u_1, v_1)$. If Z is connected, then it must form an $(k + n - 1, 0)$-core, as the degree of its vertices can decrease by at most one due to the removal of a single edge. This leads to a contradiction since $k+n-1 > k$. For the disconnected case, each one of the resulting two connected subgraphs must be a potential $(m + n - 1, 0)$-core (a non-maximal $(m + n - 1, 0)$-core) since the degree of a vertex can reduce by at most one in each component. In addition, since (u_1, v_1) is the only edge between the two disconnected components, the vertices must still have at least $k + n - 1$ in-neighbors in their respective components. One of these components must contain v, which is again a contradiction.

Next, we prove the edge deletion case. Assume $k_{max}(v)$ is decreased by n after (u_1, v_1) is removed, where $n > 1$. Inserting (u_1, v_1) back into the graph increases $k_{max}(v)$ by n, which leads to a constriction.

Proof of Theorem 4.2. We first prove the insertion case. Assume that $k_{max}(u_1)$ increases by 1 by Theorem 4.1. Then we have $(u_1, v_1) \in (k_{max}(u_1)+1, 0)$-core and consequently $v_1 \in (k_{max}(u_1)+1, 0)$-core. However, $k_{max}(v_1) < k_{max}(u_1)$ before insertion and can be most $k_{max}(u_1)$ after insertion, which implies that v_1 cannot be in $(k_{max}(u_1) + 1, 0)$-core, which leads to a contradiction.

For the deletion case, assume that $k_{max}(u_1) = n$ decreases by 1. Inserting (u_1, v_1) back to the graph should increase $k_{max}(v_1)$ to n. Then it must be true that $(u_1, v_1) \in (n, 0)$-core and $v_1 \in (n, 0)$-core. However, this is a contradiction since $k_{max}(v_1) < k_{max}(u_1)$ and $v_1 \notin (n, 0)$-core.

If $k_{max}(v_1)$ cannot be updated, then no other vertices w in G can change their degree or the k_{max} value of their in-neighbors, meaning $k_{max}(w)$ cannot be updated as well.

Proof of Theorem 4.3. First of all, all the vertices whose k_{max} has changed should form a connected subgraph. Besides, it is noteworthy that if there is any vertex w has $k_{max}(w)$ changed due to the insertion/deletion of (u_1, v_1), then $k_{max}(v_1)$ must have increased/decreased as well, as otherwise none of the vertices that have their k_{max} values changed will have their degree changed or the k_{max} value of their neighbors changed, which is a contradiction.

For edge insertion, there are two cases. For $k_{max}(w) > k_{max}(v_1)$ case, assume $k_{max}(w)$ increases to $k_{max}(w) + 1$. We must have $(u_1, v_1) \in (k_{max}(w) + 1, 0)$-core. However, this is not possible as $k_{max}(w) > k_{max}(v_1)$, i.e., a contradiction. For $k_{max}(w) < k_{max}(v_1)$ case, assume $k_{max}(w)$ increases by 1 to $k_{max}(w) + 1$. Then we have $(u_1, v_1) \in (k_{max}(w) + 1, 0)$-core. Since $k_{max}(w) + 1 \leq k_{max}(v_1) \leq k_{max}(u_1)$ and removing (u_1, v_1) from $G \oplus (u_1, v_1)$ decreases the k_{max} of u_1 and v_1 by at most one, it must be true that $k_{max}(u_1)$ and $k_{max}(v_1)$ is larger than $k_{max}(w) + 1$. This means w is in $(k_{max}(w) + 1, 0)$-core, which is a contradiction.

We have proved that only vertices with $k_{max}(w) = k_{max}(v_1)$ can have their k_{max} value incremented, and all those vertices are connected. Hence, the proof for insertion is complete. The proof for edge deletion is similar and omitted.

Proof of Theorem 4.4. Since $ED(v, G')$ is smaller than $k_{max}(v, G)$, we cannot find $k_{max}(v, G)$ in-neighbors of v with k_{max} values no less than $k_{max}(v, G)$. Based on Definition 3.1, v cannot be in $(k_{max}(v, G), 0)$-core anymore. Hence, v will have $k_{max}(v, G)$ decremented.

Proof of Theorem 4.5. The proof is straightforward. If $PED(v, G') \leq k_{max}(v, G)$, we cannot find $k_{max}(v, G) + 1$ in-neighbors of v with k_{max} values no less than $k_{max}(v, G)+1$ in $G \oplus (u_1, v_1)$, which means we cannot increment $k_{max}(v, G)$ based on the Definition 3.1.

Proof of Theorem 4.6. Referring to Theorems 4.1, 4.2, and 4.3, (i) obviously holds. For (ii), we consider two situations. First, if there are vertices v having $k_{max}(v, G) = M - 1$ incremented to M and as a result, $k_{max}(v, G') = M$, then some edges will be inserted into $(M, 0)$-core. The insertion of these edges is caused by at least one of its endpoints w having $k_{max}(w, G)$ incremented. Second, if none of the vertices $u \in V_G$ having $k_{max}(u, G)$ incremented, no edges can be inserted to $(M, 0)$-core except for (u_1, v_1). Putting these together, we prove Theorem 4.6.

Proof of Theorem 5.1. Since each $(k, 0)$-core consists of vertices with a k_{max} value no less than k, Theorem 5.1 obviously holds.

Proof of Theorem 5.2. Based on Property 3.1 of D-core, $(k_2, l_{max}(v, k_2))$-core $\subseteq (k_1, l_{max}(v, k_1))$-core holds. Hence, if v has $l_{max}(v, k_2)$ incremented due to edge insertion of (u_1, v_1), $l_{max}(v, k_1)$ will be incremented as well. Hence, Theorem 5.2 holds.

Proof of Theorem 6.1. The proof is straightforward. Based on Theorems 4.1 and 4.3, the insertion of a single edge e can only result in the change of vertices with k_{max} value being $k_{max}(e)$, and the change is at most 1. Hence, we can insert edge groups in Theorem 6.1 while ensuring that the change in the k_{max} value of any vertex is at most 1.

Proof of Theorem 6.2. We first prove that for a vertex v with $k_{max}(v, G) = k$, $k_{max}(v, G)$ can increase by at most 1 with edge insertion. Assume $k_{max}(v, G)$ is increased by n to $k + n$, where $n > 1$. At least one of the inserted edges must be in $(k + n, 0)$-core, as otherwise, $k_{max}(v, G)$ is $k+n$ as well. Let $Z = (k+n, 0)$-core $\backslash \mathcal{E}_k$. For a vertex $u \in V_Z$, if $k_{max}(u, G) < k$, then its in-degree does not change when deleting $\mathcal{E}_k$, so $deg^{in}_Z(u) \geq k+n$. If $k_{max}(u) = k$, u can lose at most one in-neighbor with k_{max} value larger than $k_{max}(u)$ in $\mathcal{E}_k$, so $deg^{in}_Z(u) \geq k+n-1$. If $k_{max}(u) \geq k+1$, u must have at least $k+1$ in-neighbors whose k_{max} values are no smaller than $k+1$. We add the vertices with k_{max} values larger than k back to Z and denote the inducted graph as Z'. Then $Z \subseteq Z' \subseteq G$ holds. In addition, from GtoZ', u does not lose any in-neighbor with k_{max} values no smaller than $k + 1$. Hence, in Z', $deg^{in}_{Z'}(u) \geq k + 1$. Then, we can see that each vertex in Z' has at least $k + 1$ in-neighbors, which means $k_{max}(v, G) > k$. However, this contradicts with $k_{max}(v, G) = k$. Hence, $k_{max}(v, G)$ can increase by at most 1. The proof to edge deletion is similar.

Based on Theorem 4.3 and the first part of this proof, it is obvious that if $k_{max}(v, G) \neq k$, $k_{max}(v)$ cannot change. Combining all of

Algorithm 7: Construction of edge groups

Construct edge groups by $k_{max}(v)$
Input: digraph $G = (V_G, E_G)$, edges E_i to be inserted/deleted, original $k_{max}(v)$ values $\mathcal{K} = \{k_{max}(v) : \forall v \in V_G\}$
Output: edge groups $\mathcal{B}$ constructed based on Strategy 1

1 Let V be a set of empty buckets with size of $max\{k_{max}(e_i), \forall e_i \in E_i\} + 1$;
2 **for** $e_i \in E_i$ **do**
3 $V_{k_{max}(e_i)}$.push(e_i);
4 Remove empty buckets in V;
5 Let $\mathcal{B}$ be a vector;
6 **while** $|V| > 1$ ***and*** $\exists B_x, B_y \in V$ ***and*** $|x - y| > 1$ **do**
7 Let S be an edge group;
  `// V is traversed in ascending order of i`
8 **for** $B_i \in V$ **do**
9 **if** B_i *is not empty* **then**
10 **if** S *is empty* ***or*** $|k_{max}(B_i.first()) - k_{max}(S.back())| > 1$ **then**
11 $e_i \leftarrow B_i$.pop();
12 S.insert(e_i);
13 **else**
14 Remove B_i from V;
15 $\mathcal{B}$.push(S);
16 **return** $\mathcal{B}$;

Construct homogeneous edge groups
Input: digraph $G = (V_G, E_G)$, edges E_k to be inserted/deleted with the same $k_{max} = x$
Output: homogeneous edge groups V constructed based on Strategy 2

17 Let V be a vector; Let $flag \leftarrow true$;
18 **while** $flag$ **do**
19 Let $\mathcal{E}_k$ be an homogeneous edge group; Let $V_{\mathcal{E}_k}$ be a vertex set;
20 **for** *each edge* $e_i = (u_i, v_i) \in E_k$ **do**
21 **if** $\mathcal{E}_k$ *is empty* ***or*** $u_i \notin V_{\mathcal{E}_k}$ **then**
  `// w.l.o.g, we assume` $k_{max}(u_i) = x$
22 $\mathcal{E}_k \leftarrow \mathcal{E}_k \cup \{e_i\}$; $V_{\mathcal{E}_k}$.push(u_i);
23 **if** $|\mathcal{E}_k| > 1$ **then**
24 V.push($\mathcal{E}_k$);
25 **else**
26 $flag \leftarrow false$;
27 **return** V;

the above, Theorem 6.2 is proved.

B DETAILED ALGORITHMS FOR EDGE GROUP CONSTRUCTION IN SECTION 6.1

The construction of edge groups is described in Algorithm 7. For Strategy 1, the algorithm initially categorizes edges into distinct buckets based on their k_{max} and removes empty buckets (lines 1-4). Subsequently, it traverses through all the buckets and checks whether there are potential edges that can be processed in parallel based on Theorem 6.1. Specifically, it checks if there are edge buckets within V containing edges with different k_{max} (line 6). In each iteration, the algorithm selects one edge from a nonempty bucket B_i (lines 8-9). If the selected edge has different k_{max} compared to the currently picked edges (line 10), the edge is removed from its original bucket and put into the selected edges (lines 11-12). Empty buckets in V will be removed (lines 13-14).

In terms of constructing homogeneous edge groups, Algorithm 7 iterates through all edges with k_{max} being x in a while loop (line 18). For a given edge e_i (line 20) and its endpoint u_i where $k_{max}(u_i) = x$, if u_i is not already included in the collected vertices, i.e., $u_i \notin V_{\mathcal{E}_k}$ (line 21), e_i is added to the current homogeneous edge group $\mathcal{E}_k$ (line 22). This procedure is repeated until no homogeneous edge group can be generated for E_k (lines 25-26). Note that the condition $|\mathcal{E}_k| > 1$ is to avoid generating homogeneous edge groups containing one edge (lines 23-24).