Accelerating D-core Maintenance over Dynamic Directed Graphs

Xuankun Liao[†1], Qing Liu[§2], Jiaxin Jiang[*3], Byron Choi[†4], Bingsheng He[*5], Jianliang Xu[†6]

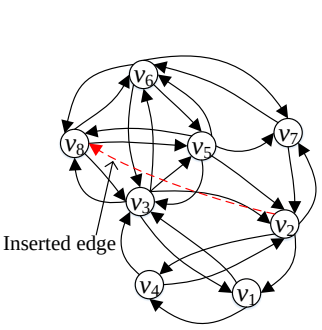
[†]*Department of Computer Science, Hong Kong Baptist University, Hong Kong, China*
[§]*College of Computer Science, Zhejiang University, Hangzhou, China*
[*]*School of Computing, National University of Singapore, Singapore*
{[1]xkliao, [4]bchoi, [6]xujl}@comp.hkbu.edu.hk, [2]qingliucs@zju.edu.cn, [3]jxjiang@nus.edu.sg,
[5]hebs@comp.nus.edu.sg

***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 smaller 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 due to sequential processing. 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 both real-world and synthetic networks demonstrate that our proposed algorithms are up to 5 orders of magnitude faster than the peeling-based method.**

I. Introduction

Directed graphs are omnipresent structures that depict large-scale entities and their relations in various fields, such as *following links* in social media platforms [1], *link relationships* in web networks [2], *money flow* in financial networks [3], and *citation relationships* in citation networks [4]. Identifying cohesive subgraphs from directed graphs has attracted considerable attention [1], [3], [5]–[7]. One well-known cohesive directed graph model is D-core, also known as (k,l)-core, in which every vertex has at least k in-neighbors and l out-neighbors [8]. 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. D-core is useful. For instance, by considering different k and l values, we can tune the relation between hubs and authorities in the related D-cores, computing groups of vertices with strong cohesion but different structures [8].

D-core decomposition is to find all D-cores for a given directed graph [3]. For example, Figure 1(b) shows all D-cores of the graph G and the modified graph G' after the insertion of edge (v_2, v_8), denoted by $G \oplus (v_2, v_8)$. Since storing all D-cores can be burdensome, the study [3] proposed storing the *anchored corenesses* of vertices. Specifically, given a vertex v and an integer 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 $\subseteq G$. 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. The D-core decomposition has a wide range of applications, such as identifying communities of strong cohesiveness in online social networks [9], [10], analyzing the structure of web graphs [2], [11], and detecting fraudsters in financial networks [12], [13].



(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_2v_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_2v_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 $G \oplus (v_2, v_8)$

Fig. 1. An example of D-core

In real-world applications, directed graphs can be highly dynamic, with frequent edge and vertex insertions and deletions [5]. For instance, in social networks, users often engage in interactions; in web graphs, new web pages and links are

constantly established; in financial networks, trading activities such as buying and selling stocks occur frequently. Consequently, D-cores in these directed graphs may also change, affecting the downstream D-core-related applications. Thus, maintaining D-cores for dynamic directed graphs is necessary. In the example of social networks, we can employ the D-core decomposition results to efficiently retrieve a user's community [9], [10]. By using a table [9] or D-Forest [10] to index vertices based on their up-to-date anchored corenesses, we can efficiently maintain the D-core-based community when the social networks update. This approach avoids the need to visit the entire updated graph and enables efficient tracking of the most recent communities.

A peeling-based method for D-core maintenance is proposed in [9], which sequentially removes vertices that violate degree constraints. Specifically, given an edge $e = (u_1, v_1)$ to be inserted or deleted, it first recomputes $(k, 0)$-cores for all possible k values by deleting vertices with the smallest in-degree one by one until the graph becomes empty. Next, for each k such that $0 \leq k \leq M$, it recomputes the updated (k, l)-cores for all possible values of l by iteratively deleting the vertices with the smallest out-degree. Here, M is a threshold value computed based on u_1 and v_1. Nevertheless, this algorithm suffers from efficiency issues. First, it will degenerate into the D-core decomposition method when M equals the maximum k value in G. Second, it handles the updated edges one by one for batch updates, making it even worse than the D-core decomposition algorithm in the worst case.

Recently, many techniques have been developed for core maintenance in different types of graphs [14]–[21]. However, these core maintenance algorithms are not applicable to D-core maintenance. Specifically, k-core maintenance algorithms maintain the *one-dimensional corenesses* of vertices based on the corenesses of their neighbors, where all the neighbors are of the same type. In contrast, D-core maintenance requires consideration of two types of neighboring vertices that influence each other and must be considered simultaneously [3]. Maintaining D-cores in dynamic directed graphs cannot be achieved accurately by considering only one type of neighbor. Furthermore, (α, β)-core maintenance in bipartite graphs differs from (k, l)-core maintenance in directed graphs. While a directed graph can be represented as a bipartite graph, establishing a correspondence between (k, l)-cores in the directed graph and (α, β)-cores in the bipartite graph, the reverse is not true. Therefore, (α, β)-core maintenance algorithms cannot be directly applied to our context. The problem of maximal D-truss maintenance (MDM), which is based on D-truss and ensures each edge forms triangles with sufficient vertices, has been recently studied [5]. Unlike D-core maintenance, which requires each *vertex* to have enough neighbors, MDM focuses on *edges*.

In this paper, we propose novel algorithms to accelerate D-core maintenance. 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 facilitate the efficient identification of vertices whose anchored corenesses require updating. For example, consider an updated edge (u_1, v_1) in a directed graph G. Assuming $k_{max}(v_1, G) = k \leq k_{max}(u_1, G)$, we find that only vertices w, for which $k_{max}(w, G)$ equals k and that are reachable from v_1 through a set of vertices z with $k_{max}(z, G) = k$ *may* have $k_{max}(w, G)$ updated. Besides, $|k_{max}(w, G') - k_{max}(w, G)| \leq 1$ holds. Here, $k_{max}(w, G)$ denotes the maximal k of vertex w such that a non-empty $(k, 0)$-core contains w in G; G' is the updated graph with $E_{G'} = E_G \cup \{(u_1, v_1)\}$ and $V_{G'} = V_G$.[1] A similar result holds for $l_{max}(v, k, G)$ of each vertex v.

Based on the theoretical findings, we introduce a two-phase local-search-based algorithm for D-core maintenance with single-edge updates, which maintains *two-dimensional* anchored corenesses in directed graphs. 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, the 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 performance further.

In practical scenarios, multiple edges may be inserted or deleted at the same time. A straightforward approach would be to process these edges individually. However, this method is inefficient, as the running time increases linearly to the number of updated edges. To enhance 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 leverages the concept of H-index [22] to iteratively calculate $k_{max}(v, G)$ or $l_{max}(v, k, G)$ for each vertex $v \in V_G$ 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 grouped edges, we can more precisely initialize the vertex values to be close to the updated ones, thereby enhancing efficiency.

We summarize the main contributions as follows.

- We systemically study the problem of D-core maintenance and conduct a thorough analysis to identify the 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.

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

In addition, two edge-grouping strategies are proposed to enhance efficiency.
- We conduct extensive empirical evaluation to validate the efficiency of our algorithms for D-core maintenance.

The rest of this paper is organized as follows. Section II reviews related works. Section III presents the formal definition of the studied problem. Sections IV and V propose the local-search-based algorithm and the H-index-based algorithm, respectively. Experimental results are reported in Section VI. Finally, Section VII concludes the paper.

II. 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 [23]. 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 [24]. Numerous efficient algorithms have been devised to address core decomposition in undirected graphs, including peeling-based approaches [25]–[27], disk-based methods [26], [27], semi-external methods [28], parallel algorithms [29], [30], and distributed solutions [31]–[33].

Moreover, core decomposition has been studied for various types of graphs, including weighted graphs [34], uncertain graphs [35], [36], bipartite graphs [20], temporal graphs [37], [38], and heterogeneous information networks [39]. Additionally, previous work has explored the core decomposition problem in directed graphs based on the D-core model [8], and efficient peeling-based algorithms and distributed algorithms for D-core decomposition have been developed [3], [9]. 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 corenesses of vertices. Sariyuce et al. [14], [15] demonstrated that the insertion/deletion of a single edge can cause a vertex's corenesses to change by at most one, based on which efficient incremental core maintenance algorithms are proposed. Similar conclusions have also been presented in [40]. Note that the dynamic k-core algorithms [14], [15] update the corenesses of vertices by visiting a local subgraph near the inserted or removed edge. Our algorithms differ from previous works in (i) the non-trial extension of updates from one-dimensional undirected corenesses to two-dimensional anchored corenesses, and (ii) the novel design of the H-index-based algorithm for batch updates. Building upon the findings of k-core maintenance, various parallel techniques that concurrently process batches of updated edges for core maintenance have been proposed [41]–[44]. In addition, Liu et al. [45] introduced an approximate algorithm to handle core maintenance with batch updates. Moreover, a distributed core maintenance algorithm has been proposed in [46]. [16]–[18] 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 [47], colorful h-star core [48], hierarchical core [49], hypercore [50]–[52], and (α, β)-core [19]–[21].

In summary, (i) most previous works on core maintenance fail to consider the difference between in-neighbors and out-neighbors, making them 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 needed.

III. Problem Formulation

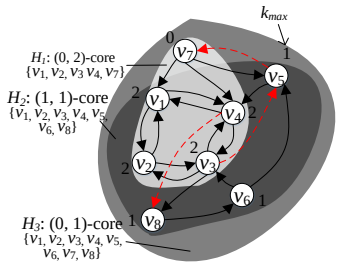
In this section, we formulate the problem of D-core maintenance. We consider a directed, unweighted simple graph $G = (V_G, E_G)$, where V_G and E_G represent the sets of vertices and edges, respectively. For brevity, we refer to a directed graph as a `digraph`. Each directed edge $e = (u, v) \in E_G$ 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., $deg_G^{in}(v) = |N_G^+(v)|$; (2) v's out-degree is the number of v's out-neighbors in G, i.e., $deg_G^{out}(v) = |N_G^-(v)|$; (3) v's degree is the sum of its in-degree and out-degree, i.e., $deg_G(v) = deg_G^{in}(v) + deg_G^{out}(v)$. Based on the in-degree and out-degree, we give the definition of D-core as follows.

Definition 1: **(D-core [8]).** 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 $\forall v \in V_H$, $deg_H^{in}(v) \geq k$ and $deg_H^{out}(v) \geq l$.

According to Definition 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 (k, l), at most one D-core exists in G. Additionally, D-core has a *partial nesting* property.

Property 1: **Partial Nesting [3].** 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 update 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.



(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

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

Definition 2: **(Anchored Coreness [3]).** 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 and v in $k_{max}(v,G)$ and $l_{max}(v,k,G)$ when the context is clear. We denote the set of anchored corenesses $\Phi(v)$ for each vertex $v \in V_G$ as $\Phi(G)$, i.e., $\Phi(G) = \bigcup_{\forall v \in V_G} \Phi(v)$. 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.

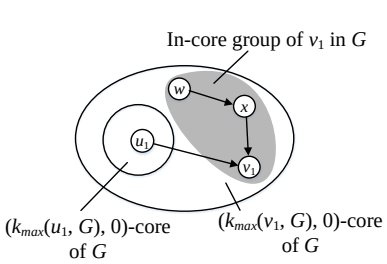
Definition 3: **(D-core Decomposition [3]).** Given a digraph G, D-core decomposition is to compute the anchored corenesses $\Phi(v)$ for each vertex v in G.

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 $V_{G'} = V_G$.

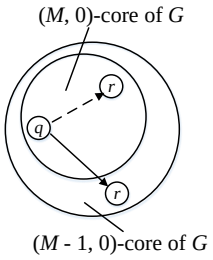
Example 1: Consider the digraph G in Figure 2. After inserting edges $E_i = \{(v_5, v_0), (v_4, v_8), (v_3, v_5)\}$, the anchored corenesses of vertices in G are updated. The updated anchored corenesses are highlighted in red in Figure 2(b).

It is noteworthy that since vertex insertion and deletion can be simulated by a sequence of edge insertions and deletions [47], we focus on edge insertions/deletions in this paper.

Boundedness Discussion of D-core Maintenance. We analyze the unboundedness of the D-core maintenance problem as follows. We denote the set of vertices whose anchored corenesses change by CHANGED. An incremental algorithm $\mathcal{A}$ is bounded if its cost is a polynomial function of $||\text{CHANGED}||_c$, where $||\cdot||_c$ is the size of c-hop neighbors for a positive integer c [5], [53]. The problem of D-core maintenance is bounded if there exists a bounded $\mathcal{A}$; otherwise, it is unbounded. Due to the space limitation, we give a proof sketch here. The detailed proof can be found in [54]. First, we prove the cost of maintaining $l_{max}(k)$ for all possible values of k is determined by the sum of the size of the visited subgraph for maintaining $l_{max}(k')$ with a specific k'. Then, we prove this sum is not bounded by $||\text{CHANGED}||_c$. Hence, D-core maintenance problem is unbounded.



(a) Illustrating Theorems 1 and 2



(b) Illustrating Theorem 4

Fig. 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))

IV. Algorithms for Single-edge Updates

In this section, we propose a local-search-based D-core maintenance algorithm to handle single-edge updates. The algorithms for the edge insertion and deletion are presented in Sections IV-A and IV-B, respectively. Then, we propose several non-trivial optimizations to accelerate the algorithms in Section IV-C.

A. 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 a digraph G and an inserted edge (u_1, v_1), $\forall v \in V_G$, the general idea is to first update $k_{max}(v,G)$, then update $l_{max}(v,k,G)$ with $k \in [0, min\{k_{max}(u_1, G'), k_{max}(v_1, G')\}]$ for a fixed k. The algorithm framework is outlined in Algorithm 1, which consists of two phases: 1) updating $k_{max}(v,G)$; 2) updating $l_{max}(v,k,G)$. Updating $k_{max}(v,G)$ and $l_{max}(v,k',G)$ with a specific k' shares the same *core idea*: first computing a set of candidate vertices using local search and then iteratively removing the vertices that do not meet specific degree constraints from the candidates. Finally, the remaining candidates w will have $k_{max}(w,G)$ or $l_{max}(w,k',G)$ updated.

Phase I: Updating $k_{max}(v,G)$. To update $k_{max}(v,G)$ for vertices $v \in V_G$, we introduce novel definitions and theorems to identify vertices v that necessitate updating $k_{max}(v,G)$ due to a single-edge update. Due to space limitations, the proofs of all theorems can be found in the technical report [54].

Given an inserted edge $e = (u_1, v_1)$, we show (i) how e affects the k_{max} values of vertices in G, and (ii) the change range of k_{max} for vertices in G.

Theorem 1: Given a digraph $G = (V_G, E_G)$ and an inserted edge $(u_1, v_1) \notin E_G$. Let $G' = G \oplus (u_1, v_1)$.

(i) If $k_{max}(u_1, G) \geq k_{max}(v_1, G)$, $k_{max}(v_1, G)$ may be updated; otherwise if $k_{max}(u_1, G) < k_{max}(v_1, G)$, $k_{max}(v, G)$

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 $\Phi(G')$

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

will not be updated for any vertex $v \in V_G$.

(ii) $\forall v \in V_G$, $k_{max}(v, G') - k_{max}(v, G) \leq 1$.

Theorem 1 shows that given an inserted 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 its $k_{max}(v_1, G)$ updated. Here, the condition of $k_{max}(u_1, G) \geq k_{max}(v_1, G)$, and the direction 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 its $k_{max}(v, G)$ updated. Additionally, the change range of $k_{max}(v, G)$ for each vertex v in G' is at most 1 by Theorem 1. Figure 3(a) shows an example where $k_{max}(v_1, G)$ may be updated if (u_1, v_1) is inserted. Note that since $k_{max}(u_1, G) > k_{max}(v_1, G)$, the $(k_{max}(u_1, G), 0)$-core is a subgraph of the $(k_{max}(v_1, G), 0)$-core, as indicated by the two nested ovals.

Next, we show that only the vertices reachable from v_1 (assuming $k_{max}(u_1, G) \geq k_{max}(v_1, G)$) and satisfying certain constraints may have their k_{max} changed.

Definition 4: **(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 $k_{max}(x, G) = k_{max}(v, G)$.

We denote the in-core group of v as $\mathcal{C}^{in}(v)$. For example, $\mathcal{C}^{in}(v_2)$ 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.

Theorem 2: Given a digraph $G = (V_G, E_G)$ and an inserted edge $(u_1, v_1) \notin E_G$. Let $G' = G \oplus (u_1, v_1)$. If $\{k_{max}(u_1, G) \geq k_{max}(v_1, G)\} \wedge \{k_{max}(v_1, G') \neq k_{max}(v_1, G)\}$, only vertices $v \in \mathcal{C}^{in}(v_1)$ may have $k_{max}(v, G)$ updated.

Besides the vertex v_1 mentioned in Theorem 1, Theorem 2 reveals that vertices $v \in \mathcal{C}^{in}(v_1)$ may also have $k_{max}(v, G)$ changed with single-edge insertions. 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 $k_{max}(v_1, G)$ is updated. Inspired by the notations of MCD and PCD in [14], we then introduce two useful definitions and corresponding theorems to prune the search space retrieved by Theorem 2, based on the 1-hop neighbors of vertices.

Definition 5: **(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 $k_{max}(u, G) \geq k_{max}(v, G)$. We define the number of exceptional in-neighbors of v as the exceptional degree of v, denoted as $ED(v, G)$.

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: $\mathcal{K} = \{k_{max}(v, G') : \forall v \in V_{G'}\}$

Let x be an integer and $x \leftarrow min\{k_{max}(u_1, G), k_{max}(v_1, G)\}$;
Let S be an empty stack and $S \leftarrow \emptyset$;
if $k_{max}(u_1, G) < k_{max}(v_1, G)$ **then**
  return $\mathcal{K} = \{k_{max}(v, G) : \forall v \in V_G\}$;
for $v \in V_{G'}$ **do**
  visited[v] $\leftarrow$ *false*; deleted[v] $\leftarrow$ *false*; d[v] $\leftarrow$ 0;
  Calculate $ED(v, G')$ and $PED(v, G')$;
d[v_1] $\leftarrow PED(v_1, G')$; S.push(v_1); visited[v_1] $\leftarrow$ *true*;
while $S \neq \emptyset$ **do**
  $v \leftarrow S$.pop();
  if $d[v] > x$ **then**
    for $(v, w) \in E_{G'}$ **do**
      if $k_{max}(w, G) = x$ ***and*** $ED(w, G') > x$ ***and*** visited[w] *is false* **then**
        S.push(w); visited[w] $\leftarrow$ *true*;
        d[w] $\leftarrow$ d[w] + $PED(w, G')$;
  else
    if deleted[v] *is false* **then**
      Mark deleted[v] as *true*; decrease d[w] for $w \in N^-_G(v)$ if $k_{max}(w, G) = x$; recursively remove w if d[w] $= x$;
for $v \in V_{G'}$ **do**
  if deleted[v] *is false* ***and*** visited[v] *is true* **then**
    $k_{max}(v, G') \leftarrow k_{max}(v, G) + 1$;
return $\mathcal{K} = \{k_{max}(v, G') : \forall v \in V_{G'}\}$;

Definition 6: **(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. Intuitively, for a vertex v, if $PED(v, G')$ is smaller than $k_{max}(v, G)$, the value of $k_{max}(v, G)$ cannot be incremented with edge insertion. Formally, we have:

Theorem 3: Given a digraph $G = (V_G, E_G)$ and an inserted edge $(u_1, v_1) \notin E_G$. Let $G' = G \oplus (u_1, v_1)$. $\forall v \in V_G$, if $PED(v, G') \leq k_{max}(v, G)$, $k_{max}(v, G') = k_{max}(v, G)$.

Theorem 3 facilitates to prune vertex v by comparing $k_{max}(v, G)$ with $PED(v, G')$. Based on the theoretical analysis, how to maintain k_{max} of vertices is outlined in Algorithm 2. Given an inserted edge (u_1, v_1), the core idea is to compute the candidate vertices v that may have their $k_{max}(v, G)$ incremented using DFS, and iteratively remove the vertices that do not have enough in-neighbors in the updated $(x, 0)$-core, where $x = min\{k_{max}(u_1, G),$

$k_{max}(v_1, G)\}$. It starts with initialization and calculation of $ED(v, G')$ and $PED(v, G')$ for each $v \in V_{G'}$ (lines 1-7), during which $\mathrm{d}[v]$ (potential in-degree of v in $(x, 0)$-core of G') will be initialized to 0. If $k_{max}(u_1, G) < k_{max}(v_1, G)$, no vertices v in G' will have $k_{max}(v, G)$ updated based on Theorem 1. Hence, Algorithm 2 will stop the maintenance (lines 3-4). Otherwise, a DFS-based search is ignited from the vertex v at the top of the stack (lines 10-11). If v and its unvisited out-neighbor w have the potential to have $k_{max}(v, G)$ and $k_{max}(w, G)$ incremented based on Theorems 3 and 2, w will be put into the stack for further check (lines 12-15). If v cannot be in $(k_{max}(v, G) + 1, 0)$-core based on Theorem 3, a recursive deletion will be started from v, during which all the out-neighbors w of v with $k_{max}(w, G) = x$ will have $\mathrm{d}[w]$ decremented by 1 (lines 16-18). If $\mathrm{d}[w]$ is x after the decrement and w is not marked as deleted, a new deletion will be started from w (lines 17-18, see the technical report [54] for more details). Finally, all the visited but not yet deleted vertices v will have $k_{max}(v, G)$ incremented by 1 (lines 19-21).

Example 2: 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 $k_{max}(v_6, G) = k_{max}(v_8, G) = 3$ and $ED(v_6, G') = 4 > 3$, $PED(v_8, G')$ equals to 1, which is smaller than $x = 3$. Hence, the algorithm removes v_8. 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 $k_{max}(v, G)$ incremented.

Phase II: Updating $l_{max}(v, k, G)$. With updated $\mathcal{K} = \{k_{max}(v, G') : \forall v \in V_{G'}\}$, to update $l_{max}(v, k, G)$ for different k values, we have to identify the difference between G_k and G'_k, i.e., the edge set $G'_k \setminus G_k$. Here, G_k is the $(k, 0)$-core of G and G'_k is the $(k, 0)$-core of G'. By inserting the new edges $e \in G'_k \setminus G_k$ into G_k, we can update $l_{max}(v, k, G)$ with an algorithm similar to Algorithm 2. Note that we only consider edges in G'_k when updating $l_{max}(v, k, G)$, and we use the $l_{max}(k)$ value of the out-neighbors of vertex v to update $l_{max}(v, k, G)$.

We then present the theorems that reveal how the structure of G_k for different k values will be updated with $\{k_{max}(v, G') : \forall v \in V_{G'}\}$. Based on Theorems 1 and 2, consider a digraph G and an inserted edge (u_1, v_1). Let $M = min\{k_{max}(u_1, G'), k_{max}(v_1, G')\}$.[2] The update of (u_1, v_1) has no effect on $(k, 0)$-cores for $k \geq M + 1$ [9]. Furthermore, it is obvious that no other edges but (u_1, v_1) will be inserted into $(k, 0)$-cores with $k \leq M - 1$. Next, we present how the $(k, 0)$-core with $k = M$ change.

Theorem 4: Given a digraph $G = (V_G, E_G)$ and an inserted edge $(u_1, v_1) \notin E_G$. Let $G' = G \oplus (u_1, v_1)$ and $M = min\{k_{max}(u_1, G'), k_{max}(v_1, G')\}$. If some vertices $V_w \subset V_{G'}$ 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 shows how the structure of $(k, 0)$-core with $k = M$ 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', and then r will be contained in $V_{G'_M}$.

Based on Theorem 4, the algorithm for $l_{max}(v, k, G)$ maintenance can be found in the technical report [54]. The key step of the $l_{max}(v, k, G)$ maintenance algorithm, i.e., maintaining $l_{max}(v, k', G)$ with a specific k', is similar to Algorithm 2. Specifically, given an inserted edge $e = (u_1, v_1)$, assume $M = min\{k_{max}(u_1, G'), k_{max}(v_1, G')\}$, the $l_{max}(v, k, G)$ maintenance algorithm inserts (u_1, v_1) into $(i, 0)$-cores with $0 \leq i \leq M - 1$ and maintains $l_{max}(k)$ values with an algorithm similar to Algorithm 2. 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. Note that the maintenance of $l_{max}(k)$ for different k values ($0 \leq k \leq M - 1$) is independent of each other, and we allocate a thread to each k for the parallel maintenance of the corresponding $l_{max}(k)$ to enhance the efficiency.

Complexity Analysis. We analyze the complexity of Algorithms 1 and 2. Algorithm 1 maintains k_{max} and $l_{max}(k)$ of vertices sequentially. Algorithm 2 maintains k_{max} of vertices, which first calculates $ED(v, G')$ and $PED(v, G')$ for each vertex v, taking $O(|E_{G'}|)$ time. Then, it performs a DFS search on vertices v with $\mathrm{d}[v]$ larger than x and removes vertices v if $\mathrm{d}[v] \leq x$, which traverses the whole graph in the worst case. Hence, Algorithm 2 has a time complexity of $O(|E_{G'}|)$. In addition, since M is bounded by $E_{G'}^{0.5}$ [9], and $l_{max}(k)$ maintenance for a specific k costs $O(|E_{G'}|)$ time, the maintenance of $l_{max}(k)$ costs $O(|E_{G'}^{1.5}|)$ time. Combining them together, Algorithm 1 takes $O(|E_{G'}^{1.5}|)$ time. The space complexities of Algorithms 1 and 2 are $O(|E_{G'}|)$. Since M is small in real-world networks, as verified in the experiments, Algorithm 1 has a time complexity close to $O(|E_{G'}|)$ and is practically efficient.

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

The theoretical results for edge deletions are similar to that of edge insertions, except for how to prune vertex v based on $ED(v, G')$. Specifically:

Theorem 5: Given a digraph $G = (V_G, E_G)$ and a deleted edge $(u_1, v_1) \in E_G$. Let $G' = G \ominus (u_1, v_1)$. $\forall v \in V_G$, if $ED(v, G') < k_{max}(v, G)$, $k_{max}(v, G') < k_{max}(v, G)$.

The algorithm handling a single-edge deletion also maintains $k_{max}(v, G)$ and $l_{max}(v, k, G)$ in a two-stage fashion

[2] It is stated in [9] 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 3: 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: $\mathcal{K} = \{k_{max}(v, G') : \forall v \in V_{G'}\}$

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 Find $\mathcal{C}^{in}(r)$ using BFS and store it in H, calculate $deg^{in}_H(w)$ for each $w \in V_H$ and store them in d;
8 **else**
9 Find $\mathcal{C}^{in}(u_1) \cup \mathcal{C}^{in}(v_1)$ using BFS and store it in H; calculate $deg^{in}_H(w)$ for each $w \in V_H$ and store them in d;
10 Sort vertices in H in increasing order w.r.t. d;
11 **for** $v \in V_H$ **do**
12 **if** $\mathrm{d}[v] < x$ **then**
13 $k_{max}(v, G') \leftarrow k_{max}(v, G) - 1$;
14 **for** $(v, w) \in E_H$ **do**
15 **if** $\mathrm{d}[w] > \mathrm{d}[v]$ **then**
16 $\mathrm{d}[w] \leftarrow \mathrm{d}[w] - 1$;
17 Resort vertex in V_H;
18 **else**
19 **Break**;
20 **return** $\mathcal{K} = \{k_{max}(v, G') : \forall v \in V_{G'}\}$;

as Algorithm 1. The maintenance of $k_{max}(v, G)$ for each v under a single-edge deletion is reported in Algorithm 3. The core idea is to compute the candidate vertices v that may have their $k_{max}(v, G)$ updated using BFS and store them in H first, and then apply a process very similar to the core decomposition algorithm [25] on H. The algorithm first does the initialization (lines 1-3). If $k_{max}(u_1, G) < k_{max}(v_1, G)$, Algorithm 3 stops the maintenance based on Theorem 1 (lines 4-5). Then, it collects candidates v that may have their $k_{max}(v, G)$ decremented (i.e., vertices in in-core group of r based on Theorems 1 and 2) using BFS (lines 6-9). Finally, a decomposition-like procedure is performed to update k_{max} for the collected vertices (lines 10-19) [25]. The $l_{max}(k)$ maintenance algorithm for edge deletions follows a structure similar to that of the $l_{max}(k)$ maintenance algorithm for edge insertions, and we omit the details for the sake of simplicity.

Example 3: Reconsider the digraph G in Figure 1, but assume that (v_2, v_8) is the edge to be deleted. We illustrate Algorithm 3. The algorithm first chooses v_8 as r since $k_{max}(v_2, G) = k_{max}(v_8, G)$, and calculates H and d. 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 $\mathrm{d}[v]$ larger than 3, no vertices will have $k_{max}(v, G)$ decremented.

Complexity Analysis. We analyze the complexity of Algorithm 3. It first finds the in-core group of r by BFS and calculates $\mathrm{d}[v]$ of each vertex v in it. This takes $O(|E_G|)$ time. Then, this algorithm iteratively removes vertices v without enough $\mathrm{d}[v]$ (i.e., $\mathrm{d}[v] < M$) from H, which also takes $O(|E_G|)$ time. Therefore, Algorithm 3 has a time complexity of $O(|E_G|)$. The space complexity is $O(|E_G|)$.

C. Optimizations

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

Optimization 1: Pruning identical $(k, 0)$-cores.
The $l_{max}(v, k, G)$ maintenance algorithm inserts (u_1, v_1) into $(k, 0)$-cores ($0 \leq k \leq M - 1$) 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 6: Consider a digraph G, $k_{max}(v, G)$ 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 $|\mathcal{I}| \leq |V_G|$. Then, for two adjacent integers I_l and I_u with $I_l - I_u > 1$, the $(k, 0)$-cores with $I_l < k \leq I_u$ have identical structures. If $I_1 \neq 0$, then $(k, 0)$-cores with $0 \leq k \leq I_1$ 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 $k_{max}(v, G) = 2$ or $k_{max}(v, G) = 3$. Hence, $(0, 0)$-core, $(1, 0)$-core, and $(2, 0)$-core have identical structures. The key idea of applying Optimization 1 is that we only insert edges into $(k, 0)$-cores with $k \in \mathcal{I}$ and maintain corresponding $l_{max}(k)$. Specifically, in the line 3 of Algorithm 1, instead of maintaining $l_{max}(v, k, G)$ for $k \in [0, min\{k_{max}(u_1, G'), k_{max}(v_1, G')\}]$, we only consider $l_{max}(v, k, G)$ for $k \in \mathcal{I}$. 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 $l_{max}(v, 0, G') = l_{max}(v, 1, G') = l_{max}(v, 2, G')$ and skip the maintenance of $l_{max}(v, 0, G)$ and $l_{max}(v, 1, G)$.

Optimization 2: Skipping redundant $l_{max}(v, k, G)$ maintenance.
Given four integers k_{low}, k_1, k_2, and k_{high} where $k_{low} \leq k_1 \leq k_2 \leq k_{high}$ holds, the rationale behind Optimization 2 is that, for a vertex v with an identical $l_{max}(v, k, G)$ across a set of consistent $k \in [k_{low}, k_{high}]$, if v has $l_{max}(v, k_2, G)$ incremented, then its $l_{max}(v, k_1, G)$ will definitely be incremented as well. The reason is that $(k_2, 0)$-core $\subseteq$ $(k_1, 0)$-core based on Property 1. Specifically,

Theorem 7: Given a digraph G, an inserted edge (u_1, v_1), four integers $k_{low} \leq k_1 \leq k_2 \leq k_{high}$, and a vertex v with a set of identical $l_{max}(v, k, G)$ across consistent k values, where $k \in [k_{low}, k_{high}]$. If v has $l_{max}(v, k_2, G)$ incremented by 1, where $k_2 \in [k_{low}, k_{high}]$, we can directly increment all the $l_{max}(v, k_1, G)$ with $k_{low} \leq k_1 \leq k_2$ by 1.

By maintaining $l_{max}(v, k, G)$ for each $v \in V_{G'}$ in the *descending* order of k, we can fully utilize Optimization 2. Specifically, in line 3 of Algorithm 1, when maintaining $l_{max}(v, k, G)$ with different k values, if we find vertices v that satisfy the constraint in Theorem 7, we directly increment all $l_{max}(v, k_1, G)$ with $k_{low} \leq k_1 \leq k_2$ by 1, rather than checking whether $l_{max}(v, k_1, G)$ should be incremented by cumbersome DFS search, thereby enhancing performance. For example, given a vertex v and $\Phi(v) = \{(0, 3), (1, 2), (2, 2), (3, 2)\}$, we can see that for $k \in [1, 3]$, $l_{max}(v, k, G)$ are the same. Assume $l_{max}(v, 3, G)$ is incremented to 3 with a single-

edge insertion. We can directly increase $l_{max}(v,1,G)$ and $l_{max}(v,2,G)$ by one as well according to Theorem 7. 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 the $l_{max}(v,k,G)$ maintenance algorithm, we process each different $(k,0)$-core from scratch. However, based on Property 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 the $l_{max}(v,k,G)$ maintenance algorithm, we can use the adjacent lists from the previous $(k_2,0)$-core and compute new adjacent lists based on new vertices V_N in the current $(k_1,0)$-core $(k_2 > k_1)$, i.e., $V_N = V_{(k_1,0)-\text{core}} \setminus V_{(k_2,0)-\text{core}}$.

V. 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 and its extension for batch deletions. Finally, we present the overall H-index-based D-core maintenance algorithm.

A. 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 [54]. Those strategies are applicable to both k_{max} and $l_{max}(k)$, 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., $k_{max}((u,v)) = min\{k_{max}(u,G), k_{max}(v,G)\}$.

Strategy 1: Group by $k_{max}((u,v))$. As Theorem 1 suggests, the insertion or deletion of an edge can lead to a change in $k_{max}(v,G)$ 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 8: Given a digraph G and a collection of edges E_i to be inserted/deleted. Consider edge groups consisting of edges in E_i, which are denoted as as $\mathcal{B} = \{B_1, B_2, \ldots, B_j\}$. If (i) 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; and (ii) for any two edge groups B_x and B_y, where $0 \le x \le j$ and $0 \le y \le j$, $B_x \cap B_y = \emptyset$. Then, the insertion/deletion of any edge group $B_i \subset \mathcal{B}$ leads to a change in $k_{max}(v,G)$ of any vertex $v \in V_G$ by at most 1.

Theorem 8 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 numerous edges have the same k_{max} value, 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.

TABLE I
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	$ED(v,G)$	2	2	2	2	1	1	0	1
	$PED(v,G)$	0	0	0	0	0	0	0	1
$G' = G \oplus \{(v_3,v_5),(v_4,v_8)\}$	$ED(v,G')$	2	2	2	2	1	1	0	**2**
	$PED(v,G')$	0	0	0	0	0	0	0	**2**
$G'' = G' \oplus \{(v_5,v_7)\}$	$ED(v,G'')$	2	2	2	2	**2**	1	**1**	**2**
	$PED(v,G'')$	0	0	0	0	**1**	**1**	**1**	**0**

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 $(k_{max}(v,G),0)$-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.

Definition 7: **(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 $e_1 \cap e_2 = w$, $k_{max}(w,G) > k$.

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 $k_{max}(v,G)$ of each vertex v.

Theorem 9: Given a digraph G and a homogeneous edge group $\mathcal{E}_k$ to be inserted/deleted, $\forall v \in V_G$, only v with $k_{max}(v,G) = k$ may have $k_{max}(v,G)$ changed, and this change is no more than 1.

Example 4: 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 calculated 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 $B_2 = \{(v_5,v_7)\}$.

B. The H-index-based $k_{max}(v,G)$ 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 [22]. 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,

TABLE II
An illustration of D-core maintenance using Algorithm 4 on graph G in Figure 2.

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

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 8: **(n-order in-H-index [3]).** Given a vertex v in G, the n-order in-H-index of v, denoted by $\text{iH}_G^{(n)}(v)$, is defined as

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

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

Theorem 10 (Convergence [3]):

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

Theorem 10 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 IV and the strategies in Section V-A to enhance the efficiency.

Vertex value initialization for a batch insertion. Based on Theorems 1, 2, and 3, given an inserted edge (u_1,v_1), if $k_{max}(u_1,G) \geq k_{max}(v_1,G)$, only vertices v in $\mathcal{C}^{in}(v_1)$ 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 1. In addition, Theorems 8 and 9 show that we can find a group of edges and insert them simultaneously while making sure that $k_{max}(v,G)$ 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 V-A as follows:

1) For vertex $v \in \bigcup_{\{(u_1,v_1)\}\in B_i} \mathcal{C}^{in}(v_1)$, 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 $k_{max}(v,G)$ updated. The details for the H-index-based algorithm maintaining k_{max} with a batch insertion are outlined in Algorithm 4. The core idea is to compute the candidate vertices v that may have their $k_{max}(v,G)$ updated with the proposed vertex value initialization strategy first, and then apply the concept of n-order in-H-index to calculate $k_{max}(v,G')$ for the affected vertices $v \in V_{G'}$. The algorithm first does the initialization for each vertex v (lines 1-4). Then, it calculates r (lines 5-8),

Algorithm 4: 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 V-A, $\mathcal{K} = \{k_{max}(v,G) : v \in V_G\}$
Output: $\mathcal{K}' = \{k_{max}(v,G') : \forall v \in V_{G'}\}$

```
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 (u1,v1) ∈ B_i do
6      Let r be a vertex and r ← v1;
7      if k_max(u1,G) < k_max(v1,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'};
```

assigns the initial vertex value, and marks vertices as active when necessary based on our vertex value initiation strategy (lines 9-13). Note that since no graph updates or k_{max} updates will be made to G and vertices in G in lines 5-13, no vertices will have their ED and PED updated in lines 5-13. 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 $\mathcal{H}(I) < \text{iH}(v)$, the algorithm 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.

Edge deletion extension of Algorithm 4. We further discuss how to extend the H-index-based algorithm for edge insertions in Algorithm 4 to the edge deletion case. The extension can be easily achieved by modifying the vertex initialization in Section V-B. Specifically, based on Theorems 1, 2 and 5, we should initialize v's vertex value as $k_{max}(v,G)$ for vertex $v \in \bigcup_{\{(u_1,v_1)\}\in B_d} \mathcal{C}^{in}(v_1)$, 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.

Example 5: We use digraph G in Figure 2 to illustrate Algorithm 4, whose calculation process is shown in Table II.

Table I 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 $k_{max}(v_8,G)+1$, i.e., $\text{iH}^{(0)}(v_8)=2$, as $PED(v_8,G')=2 > k_{max}(v_8,G)=1$. Then, Algorithm 4 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, $k_{max}(v_8,G)$ is updated as 2. Similarly, $k_{max}(v_7,G)$ is updated as 1 with the insertion of $\{(v_5,v_7)\}$.

C. The Overall H-index-based Algorithm

We omit the details for the complete H-index-based algorithm and put them into the technical report [54] due to space limitations. Similar to Algorithm 1, the complete H-index-based algorithm for a batch insertion also maintains k_{max} and $l_{max}(k)$ sequentially. Given a batch of inserted edges E_i, the algorithm first constructs the edge groups $\mathcal{B}$ for maintenance of $\mathcal{K}=\{k_{max}(v,G) : \forall v \in V_G\}$ based on Strategies 1 and 2. Next, for each generated group B, the algorithm maintains $\mathcal{K}$ using Algorithm 4, records the maximum k_{max} of edges in $\mathcal{B}$ with M_{max} , and inserts B into G. The edges in E_i that cannot be processed in parallel are handled sequentially, and the maximum k_{max} of edges in E_i will be recorded at the same time if it is larger than M_{max}. The $l_{max}(k)$ maintenance for $k \in [0, M_{max}]$ follows a similar procedure. Our proposed algorithms can support batch updates involving both edge insertions and deletions. The strategy is to process them separately. Specifically, we can first employ Algorithm 4 to insert the edges and update the anchored corenesses for vertices, then delete the other edges and update the corresponding corenesses with the H-index-based edge deletion algorithm.

Complexity analysis. Let the maximum in-degree $\Delta_{in} = \max_{v\in V_G} deg_G^{in}(v)$ and R_i (resp. R_d) be the number of convergence rounds required by Algorithm 4 (resp. the H-index-based edge deletion algorithm). For edge insertion, calculating ED and PED for each vertex takes $O(E_G)$ time, and vertex value initialization costs $O(|V_G|\cdot|B_i|)$ time. Updating anchored corenesses takes $O(R_i\cdot|V_G|\cdot\Delta_{in})$ time. Thus, Algorithm 4 runs in $O(|V_G|\cdot(R_i\cdot\Delta_{in}+|B_i|)+E_G)$ time. For edge deletion, calculating ED for each vertex takes $O(E_G)$ time, and vertex value initialization costs $O(|V_G|\cdot|B_d|)$ time, where B_d is the deleted edge group. Updating anchored corenesses takes $O(R_d\cdot|V_G|\cdot\Delta_{in})$ time. Thus, the H-index-based edge deletion algorithm runs in $O(|V_G|\cdot(R_d\cdot\Delta_{in}+|B_d|)+E_G)$ time. Both algorithms use $O(E_G)$ space.

VI. 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 128GB 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 III
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	54	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	253	254
Power Law	PL	2.6**M**	116.2**M**	44.63	436	433
Uniform	UD	3.0**M**	150.0**M**	50.16	56	59
Hollywood	HW	2.1**M**	228.9**M**	105.00	1,297	1,297

Table III 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. Power Law and Uniform are two generated synthetic graphs whose degree distributions follow the power-law distribution and uniform distribution, respectively.

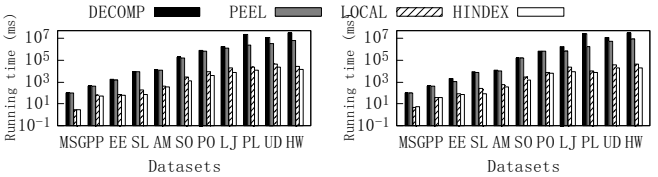
Algorithms. We compare several algorithms in our experiments.

1) **DECOMP** [9]: The state-of-the-art D-core decomposition algorithm.
2) **PEEL** [9]: 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.

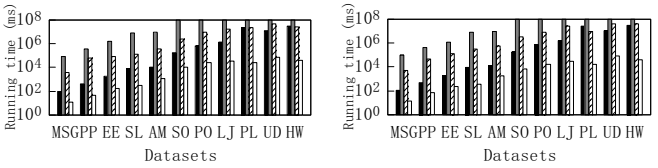
Parameters and metrics. The parameters tested in experiments include the number of threads, the number of updated edges ($|\Delta G|$), and the graph size, with default settings of 16, 1K, and 100%$\cdot|E_G|$, respectively. We randomly generate edges to be inserted/deleted. Two update modes are considered, 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) and the improvement brought by optimizations in Section IV-C (in percentage). We terminate 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 takes over 2.5 hours to process the largest dataset HW with over 200 million edges, while our methods take less

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

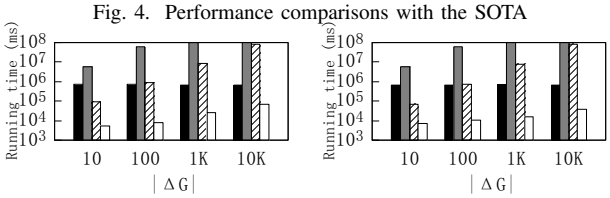


(a) Single-edge updates, deletion (b) Single-edge updates, insertion



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

Fig. 4. Performance comparisons with the SOTA

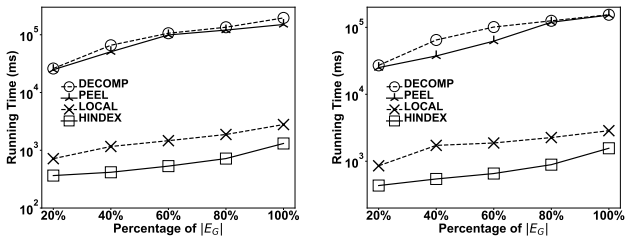


(a) PO, batch deletions (b) PO, batch insertions

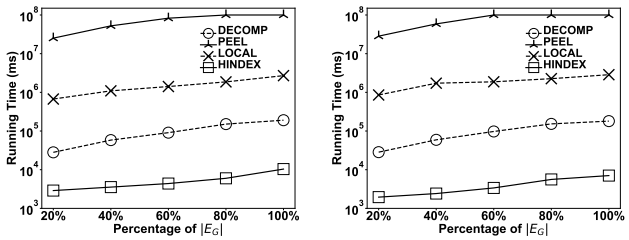
Fig. 5. Effect of the number of updated edges $|\Delta G|$

than one 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, , PL, UD, and HW, while HINDEX can finish within a minute. Besides, HINDEX outperforms PEEL by up to 20,000 times over the other datasets and is up to 700 times faster than DECOMP over all datasets. These results 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, PL, UD, 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 for batch update. As shown in Figure 5, with the increase of $|\Delta G|$, the running time of DECOMP is almost unchanged, as its cost is $O(|G \oplus \Delta G|^{1.5})$ [9], 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 when $|\Delta G| = 100$. 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, in Figure 5(b), HINDEX outperforms DECOMP up to 20 times when $|\Delta G| = 10$K. On average, HINDEX outperforms



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



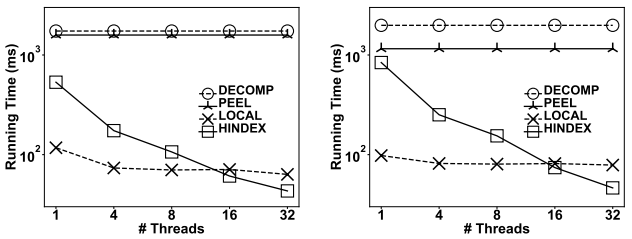
(c) SO, batch deletions (d) SO, batch insertions

Fig. 6. Effect of cardinality

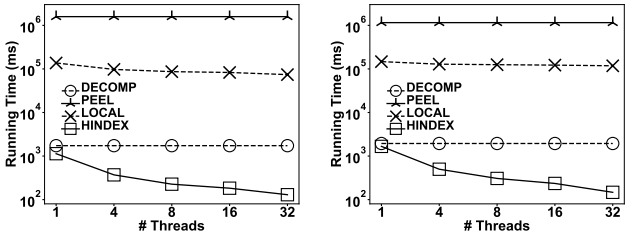
DECOMP by 63.7 and 51.2 times for edge deletions and insertions, respectively.

Exp-3: Effect of dataset cardinality. We evaluate the effect of cardinality on all the algorithms. 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 because when the dataset becomes larger, more vertices need to update the anchored corenesses, resulting in more running time. 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. In this experiment, we assess the effect of the number of threads. Figure 7 presents the results on dataset EE. 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. For instance, in Figure 7(d), HINDEX demonstrates an 11.3x speedup when the number of threads increases from 1 to 32 for batch insertion case. Additionally, HINDEX exhibits superior parallelism performance compared to LOCAL. This is because LOCAL parallelizes the computation of $l_{max}(k)$ with $0 \leq k \leq M-1$, and M is usually smaller than the number of threads, thus limiting the performance improvement from multiple threads. Here, M is $min\{k_{max}(u_1,G'), k_{max}(v_1,G')\}$ and (u_1,v_1) is the updated edge. 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 of our proposed algorithms are faster than PEEL and DECOMP for single-edge updates.

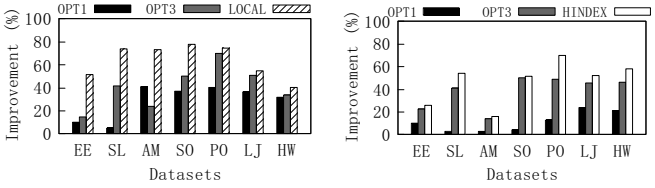


(a) EE, single-edge deletions (b) EE, single-edge insertions

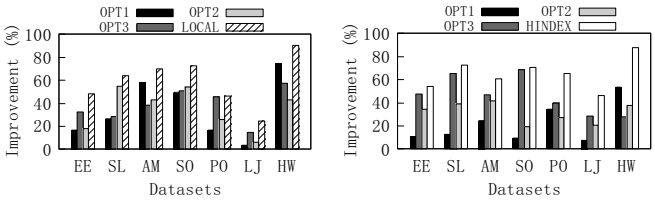


(c) EE, batch deletions (d) EE, batch insertions

Fig. 7. Effect of # threads



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

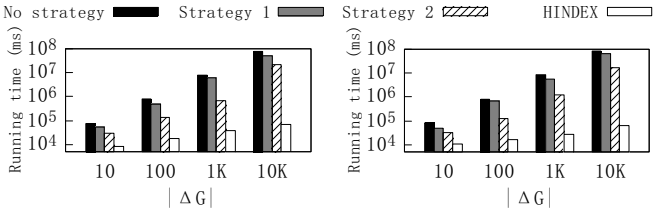


(c) **LOCAL**, single insertions (d) **HINDEX**, single insertions

Fig. 8. Effect of optimizations for single-edge updates

Exp-5: Effect of optimizations. We now show the effectiveness of three optimizations proposed in Section IV-C. It is worth noting that (1) Optimization 2 does not apply to the edge deletion case, and (2) all three optimizations can be integrated into the H-index-based algorithm. Figure 8 reports the results for single-edge updates. Since the batch updates have similar trends, we omit them due to space limitations. We observe that all three optimizations are effective. For instance, in the case of edge insertions on HW dataset, OPT1, OPT2, and OPT3 enhance the performance of the local search algorithm by up to 74%, 42%, and 57%, respectively. The three optimizations together can accelerate the local search algorithm by up to 90%. 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.

Exp-6: Effect of edge-grouping strategies. We evaluate the effect of different edge-grouping strategies for batch processing in the H-index-based algorithm. Figure 9 shows the results for batch updates. We can observe that our proposed edge-grouping strategies can efficiently identify the edges that can be simultaneously inserted into or deleted from the original graph, and thus improve the efficiency of the H-index-based algorithm for batch updates. For instance, in Figure 9(b), when $|\Delta G| = 10$K, HINDEX outperforms the H-index-based algorithm without any edge-grouping strategies up to three orders of magnitude. In addition, we can see that the performance of Strategy 2 is better than Strategy 1. It is because most of the updated edges have the same k_{max} value.



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

Fig. 9. Effect edge-grouping strategies

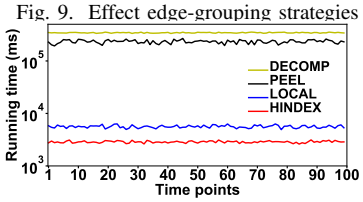


Fig. 10. Performance of handling the sliding window update mode

Exp-7: Performance of handling the sliding window update mode. Finally, we evaluate the efficiency of our algorithms for handling the sliding window update mode. Figure 10 shows the result on SO, a temporal graph consisting of interactions on Stack Overflow. The edges in SO are sorted in ascending order by timestamp. For edges with multiple timestamps, we retain only their earliest timestamp. We start at the time point of 1 and initialize the graph with the earliest $|E_G| - 100$ edges. For each slide, we insert one subsequent edge and remove the edge with the earliest timestamp. The total running time of the algorithms for edge insertion and deletion is reported. As shown in Figure 10, HINDEX and LOCAL consistently outperform DECOMP and PEEL.

VII. Conclusion

In this paper, we study the D-core maintenance problem in dynamic digraphs. To address this problem, we first introduce new theoretical findings to identify vertices that require anchored coreness updates. To handle D-core maintenance with single-edge insertions or deletions, we propose a local-search-based algorithm along with three optimizations based on the theoretical findings. To enhance the performance of batch updates, we propose an H-index-based algorithm incorporating innovative edge-grouping strategies. Theoretical analysis and extensive empirical evaluations demonstrate the efficiency of our proposed algorithms. In the future, we would like to explore D-core maintenance in *distributed* settings.

REFERENCES

[1] Q. Liu, M. Zhao, X. Huang, J. Xu, and Y. Gao, "Truss-based community search over large directed graphs," in *Proceedings of the 2020 ACM International Conference on Management of Data*, 2020, pp. 2183–2197.

[2] B. Abedin and B. Sohrabi, "Graph theory application and web page ranking for website link structure improvement," *Behaviour & Information Technology*, vol. 28, no. 1, pp. 63–72, 2009.

[3] X. Liao, Q. Liu, J. Jiang, X. Huang, J. Xu, and B. Choi, "Distributed d-core decomposition over large directed graphs," *Proceedings of the VLDB Endowment*, vol. 15, no. 8, pp. 1546–1558, 2022.

[4] D. J. D. S. Price, "Networks of scientific papers: The pattern of bibliographic references indicates the nature of the scientific research front." *Science*, vol. 149, no. 3683, pp. 510–515, 1965.

[5] A. Tian, A. Zhou, Y. Wang, and L. Chen, "Maximal d-truss search in dynamic directed graphs," *Proceedings of the VLDB Endowment*, vol. 16, no. 9, pp. 2199–2211, 2022.

[6] C. Ma, Y. Fang, R. Cheng, L. V. Lakshmanan, W. Zhang, and X. Lin, "On directed densest subgraph discovery," *ACM Transactions on Database Systems (TODS)*, vol. 46, no. 4, pp. 1–45, 2021.

[7] X. Huang, L. V. Lakshmanan, and J. Xu, "Community search over big graphs," *Synthesis Lectures on Data Management*, vol. 14, no. 6, pp. 1–206, 2019.

[8] C. Giatsidis, D. M. Thilikos, and M. Vazirgiannis, "D-cores: measuring collaboration of directed graphs based on degeneracy," *Knowledge and information systems*, vol. 35, no. 2, pp. 311–343, 2013.

[9] Y. Fang, Z. Wang, R. Cheng, H. Wang, and J. Hu, "Effective and efficient community search over large directed graphs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 31, no. 11, pp. 2093–2107, 2018.

[10] Y. Chen, J. Zhang, Y. Fang, X. Cao, and I. King, "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*, 2021, pp. 3544–3550.

[11] J. M. Kleinberg, R. Kumar, P. Raghavan, S. Rajagopalan, and A. S. Tomkins, "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, 1999, pp. 1–17.

[12] P. Irofti, A. Patrascu, and A. Baltoiu, "Quick survey of graph-based fraud detection methods," *arXiv preprint arXiv:1910.11299*, 2019.

[13] Z. Li, H. Xiong, Y. Liu, and A. Zhou, "Detecting blackhole and volcano patterns in directed networks," in *2010 IEEE International Conference on Data Mining*. IEEE, 2010, pp. 294–303.

[14] A. E. Saríyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and Ü. V. Çatalyürek, "Streaming algorithms for k-core decomposition," *Proceedings of the VLDB Endowment*, vol. 6, no. 6, pp. 433–444, 2013.

[15] A. E. Sarıyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and Ü. V. Çatalyürek, "Incremental k-core decomposition: algorithms and evaluation," *The VLDB Journal*, vol. 25, pp. 425–447, 2016.

[16] Y. Zhang, J. X. Yu, Y. Zhang, and L. Qin, "A fast order-based approach for core maintenance," in *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 2017, pp. 337–348.

[17] B. Guo and E. Sekerinski, "Simplified algorithms for order-based core maintenance," *arXiv preprint arXiv:2201.07103*, 2022.

[18] B. Guo and E. Sekerinski, "Parallel order-based core maintenance in dynamic graphs," in *Proceedings of the 52nd International Conference on Parallel Processing*, 2023, pp. 122–131.

[19] W. Bai, Y. Chen, D. Wu, Z. Huang, Y. Zhou, and C. Xu, "Generalized core maintenance of dynamic bipartite graphs," *Data Mining and Knowledge Discovery*, pp. 1–31, 2022.

[20] B. Liu, L. Yuan, X. Lin, L. Qin, W. Zhang, and J. Zhou, "Efficient (a,β)-core computation: an index-based approach," in *International World Wide Web Conference*, 2019, pp. 1130–1141.

[21] W. Luo, Q. Yang, Y. Fang, and X. Zhou, "Efficient core maintenance in large bipartite graphs," *Proceedings of the ACM on Management of Data*, vol. 1, no. 3, pp. 1–26, 2023.

[22] J. E. Hirsch. (2005) H-index. [Online]. Available: https://en.wikipedia.org/wiki/H-index

[23] S. B. Seidman, "Network structure and minimum degree," *Social Networks*, vol. 5, no. 3, pp. 269–287, 1983.

[24] F. D. Malliaros, C. Giatsidis, A. N. Papadopoulos, and M. Vazirgiannis, "The core decomposition of networks: Theory, algorithms and applications," *The VLDB Journal*, vol. 29, pp. 61–92, 2020.

[25] V. Batagelj and M. Zaversnik, "An o (m) algorithm for cores decomposition of networks," *arXiv preprint cs/0310049*, 2003.

[26] J. Cheng, Y. Ke, S. Chu, and M. T. Özsu, "Efficient core decomposition in massive networks," in *2011 IEEE 27th International Conference on Data Engineering*, 2011, pp. 51–62.

[27] W. Khaouid, M. Barsky, V. Srinivasan, and A. Thomo, "K-core decomposition of large networks on a single pc," *Proceedings of the VLDB Endowment*, vol. 9, no. 1, pp. 13–23, 2015.

[28] D. Wen, L. Qin, Y. Zhang, X. Lin, and J. X. Yu, "I/O efficient core graph decomposition at web scale," in *International Conference on Data Engineering*, 2016, pp. 133–144.

[29] N. S. Dasari, R. Desh, and M. Zubair, "Park: An efficient algorithm for k-core decomposition on multicore processors," in *2014 IEEE International Conference on Big Data*, 2014, pp. 9–16.

[30] H. Esfandiari, S. Lattanzi, and V. Mirrokni, "Parallel and streaming algorithms for k-core decomposition," in *International Conference on Machine Learning*, 2018, pp. 1397–1406.

[31] S. Aridhi, M. Brugnara, A. Montresor, and Y. Velegrakis, "Distributed k-core decomposition and maintenance in large dynamic graphs," in *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, 2016, pp. 161–168.

[32] A. Montresor, F. De Pellegrini, and D. Miorandi, "Distributed k-core decomposition," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 2, pp. 288–300, 2012.

[33] A. Mandal and M. A. Hasan, "A distributed k-core decomposition algorithm on spark," in *IEEE International Conference on Big Data*, 2017, pp. 976–981.

[34] M. Eidsaa and E. Almaas, "S-core network decomposition: A generalization of k-core analysis to weighted networks," *Physical Review E*, vol. 88, no. 6, p. 062819, 2013.

[35] F. Bonchi, F. Gullo, A. Kaltenbrunner, and Y. Volkovich, "Core decomposition of uncertain graphs," in *Proceedings of the 20th International Conference on Knowledge Discovery and Data Mining*, 2014, pp. 1316–1325.

[36] Y. Peng, Y. Zhang, W. Zhang, X. Lin, and L. Qin, "Efficient probabilistic k-core computation on uncertain graphs," in *IEEE International Conference on Data Engineering*, 2018, pp. 1192–1203.

[37] E. Galimberti, M. Ciaperoni, A. Barrat, F. Bonchi, C. Cattuto, and F. Gullo, "Span-core decomposition for temporal networks: Algorithms and applications," *ACM Transactions on Knowledge Discovery from Data*, vol. 15, no. 1, pp. 2:1–2:44, 2021.

[38] H. Wu, J. Cheng, Y. Lu, Y. Ke, Y. Huang, D. Yan, and H. Wu, "Core decomposition in large temporal graphs," in *IEEE International Conference on Big Data*, 2015, pp. 649–658.

[39] Y. Fang, Y. Yang, W. Zhang, X. Lin, and X. Cao, "Effective and efficient community search over large heterogeneous information networks," *Proceedings of the VLDB Endowment*, vol. 13, no. 6, pp. 854–867, 2020.

[40] R.-H. Li, J. X. Yu, and R. Mao, "Efficient core maintenance in large dynamic graphs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 10, pp. 2453–2465, 2013.

[41] Q.-S. Hua, Y. Shi, D. Yu, H. Jin, J. Yu, Z. Cai, X. Cheng, and H. Chen, "Faster parallel core maintenance algorithms in dynamic graphs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 6, pp. 1287–1300, 2019.

[42] H. Jin, N. Wang, D. Yu, Q.-S. Hua, X. Shi, and X. Xie, "Core maintenance in dynamic graphs: A parallel approach based on matching," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 11, pp. 2416–2428, 2018.

[43] N. Wang, D. Yu, H. Jin, C. Qian, X. Xie, and Q.-S. Hua, "Parallel algorithm for core maintenance in dynamic graphs," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 2366–2371.

[44] W. Bai, Y. Jiang, Y. Tang, and Y. Li, "Parallel core maintenance of dynamic graphs," *IEEE Transactions on Knowledge and Data Engineering*, 2022.

[45] Q. C. Liu, J. Shi, S. Yu, L. Dhulipala, and J. Shun, "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*, 2022, pp. 191–204.

[46] H. Aksu, M. Canim, Y.-C. Chang, I. Korpeoglu, and Ö. Ulusoy, "Distributed k-core view materialization and maintenance for large dynamic graphs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 10, pp. 2439–2452, 2014.

[47] Q. Liu, X. Zhu, X. Huang, and J. Xu, "Local algorithms for distance-generalized core decomposition over large dynamic graphs," *Proceedings of the VLDB Endowment*, vol. 14, no. 9, pp. 1531–1543, 2021.

[48] S. Gao, H. Qin, R.-H. Li, and B. He, "Parallel colorful h-star core maintenance in dynamic graphs," *Proceedings of the VLDB Endowment*, vol. 16, no. 10, pp. 2538–2550, 2023.
[49] Z. Lin, F. Zhang, X. Lin, W. Zhang, and Z. Tian, "Hierarchical core maintenance on large dynamic graphs," *Proceedings of the VLDB Endowment*, vol. 14, no. 5, pp. 757–770, 2021.
[50] Q. Luo, D. Yu, Z. Cai, X. Lin, G. Wang, and X. Cheng, "Toward maintenance of hypercores in large-scale dynamic hypergraphs," *The VLDB Journal*, vol. 32, no. 3, pp. 647–664, 2023.
[51] Q. Luo, D. Yu, Z. Cai, Y. Zheng, X. Cheng, and X. Lin, "Core maintenance for hypergraph streams," *World Wide Web*, vol. 26, no. 5, pp. 3709–3733, 2023.
[52] Q.-S. Hua, X. Zhang, H. Jin, and H. Huang, "Revisiting core maintenance for dynamic hypergraphs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 3, pp. 981–994, 2023.
[53] Y. Zhang and J. X. Yu, "Unboundedness and efficiency of truss maintenance in evolving graphs," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 1024–1041.
[54] X. Liao, Q. Liu, J. Jiang, B. Choi, B. He, and J. Xu, "D-core maintenance technical report," 2024. [Online]. Available: https://github.com/LIAOXuankun/Dynamic-Dcore-techreport/blob/main/Dcore-Maintenance-tech-report.pdf

APPENDIX

A. Proofs to the theorems

We provide the detailed proofs of theorems in Sections IV and V in the appendix.

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

For the deletion case, assume that $k_{max}(u_1, G) = n$ decreases by 1. Inserting (u_1, v_1) back to the graph should increase $k_{max}(v_1, G)$ 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 $k_{max}(v_1, G) < k_{max}(u_1, G)$ and $v_1 \notin (n, 0)$-core.

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

For the second part of the theorem, we first prove the insertion case. Assume that after the insertion of (u_1, v_1), $k_{max}(v, G) = k$ 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 $k_{max}(v, G)$ 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 $k_{max}(v, G)$ is decreased by n after (u_1, v_1) is removed, where $n > 1$. Inserting (u_1, v_1) back into the graph increases $k_{max}(v, G)$ by n, which leads to a constriction.

Proof of Theorem 2. 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 $k_{max}(w, G)$ changed due to the insertion/deletion of (u_1, v_1), then $k_{max}(v_1, G)$ 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 $k_{max}(w, G) > k_{max}(v_1, G)$ case, assume $k_{max}(w, G)$ increases to $k_{max}(w, G) + 1$. We must have $(u_1, v_1) \in (k_{max}(w, G) + 1, 0)$-core. However, this is not possible as $k_{max}(w, G) > k_{max}(v_1, G)$, i.e., a contradiction. For $k_{max}(w, G) < k_{max}(v_1, G)$ case, assume $k_{max}(w, G)$ increases by 1 to $k_{max}(w, G)+1$. Then we have $(u_1, v_1) \in (k_{max}(w, G) + 1, 0)$-core. Since $k_{max}(w, G) + 1 \leq k_{max}(v_1, G) \leq k_{max}(u_1, G)$ and removing (u_1, v_1) from $G \oplus (u_1, v_1)$ decreases $k_{max}(u_1, G)$ and $k_{max}(v_1, G)$ by at most one, it must be true that $k_{max}(u_1, G)$ and $k_{max}(v_1, G)$ are larger than $k_{max}(w, G) + 1$. This means w is in $(k_{max}(w, G) + 1, 0)$-core, which is a contradiction.

We have proved that only vertices with $k_{max}(w, G) = k_{max}(v_1, G)$ 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 5. 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 1, v cannot be in $(k_{max}(v, G), 0)$-core anymore. Hence, v will have $k_{max}(v, G)$ decremented.

Proof of Theorem 3. 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 1.

Proof of Theorem 4. Referring to Theorems 1, **??**, and 2, (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.

Proof of Theorem 6. Since each $(k, 0)$-core consists of vertices with a k_{max} value no less than k, Theorem 6 obviously holds.

Proof of Theorem 7. Based on Property 1 of D-core, $(k_2, l_{max}(v, k_2, G))$-core $\subseteq (k_1, l_{max}(v, k_1, G))$-core holds. Hence, if v has $l_{max}(v, k_2, G)$ incremented due to edge insertion of (u_1, v_1), $l_{max}(v, k_1, G)$ will be incremented as well. Hence, Theorem 7 holds.

Proof of Theorem 8. The proof is straightforward. Based on Theorems 1 and 2, 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 8 while ensuring that the change in the k_{max} value of any vertex is at most 1.

Proof of Theorem 9. 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 $k_{max}(u, G) = k$, u can lose at most one in-neighbor with k_{max} value larger than $k_{max}(u, G)$ in

Algorithm 5: Construction of edge groups

Construct edge groups by $k_{max}(v, G)$
Input: digraph $G = (V_G, E_G)$, edges E_i to be inserted/deleted, $\mathcal{K} = \{k_{max}(v, G) : \forall v \in V_G\}$
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 $\quad V_{k_{max}(e_i)}$.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 $\quad$ Let S be an edge group;
$\quad$ `// V is traversed in ascending order of i`
8 $\quad$ **for** $B_i \in V$ **do**
9 $\quad\quad$ **if** B_i *is not empty* **then**
10 $\quad\quad\quad$ **if** S *is empty* ***or*** $|k_{max}(B_i.first()) - k_{max}(S.back())| > 1$ **then**
11 $\quad\quad\quad\quad$ $e_i \leftarrow B_i$.pop();
12 $\quad\quad\quad\quad$ S.insert(e_i);
13 $\quad\quad$ **else**
14 $\quad\quad\quad$ Remove B_i from V;
15 $\quad$ $\mathcal{B}$.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 $\quad$ Let $\mathcal{E}_k$ be an homogeneous edge group; Let $V_{\mathcal{E}_k}$ be a vertex set;
20 $\quad$ **for** *each edge* $e_i = (u_i, v_i) \in E_k$ **do**
21 $\quad\quad$ **if** $\mathcal{E}_k$ *is empty* ***or*** $u_i \notin V_{\mathcal{E}_k}$ **then**
$\quad\quad\quad$ `// w.l.o.g, we assume` $k_{max}(u_i, G) = x$
22 $\quad\quad\quad$ $\mathcal{E}_k \leftarrow \mathcal{E}_k \cup \{e_i\}$; $V_{\mathcal{E}_k}$.push(u_i);
23 $\quad$ **if** $|\mathcal{E}_k| > 1$ **then**
24 $\quad\quad$ V.push($\mathcal{E}_k$);
25 $\quad$ **else**
26 $\quad\quad$ $flag \leftarrow false$;
27 **return** V;

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 $\quad$ Obtain $k_{max}(w, G)$ from $\mathcal{K}$ by a mapping in $O(1)$ time;
4 $\quad$ **if** $k_{max}(w, G) = x$ **then**
5 $\quad\quad$ d[w] $\leftarrow$ d[w] - 1;
6 $\quad\quad$ **if** d[w] $= x$ ***and*** deleted[w] *is* $false$ **then**
7 $\quad\quad\quad$ REMOVE ($G = (V_G, E_G)$, $\mathcal{K}$, d, deleted, x, w)

$\mathcal{E}_k$, so $deg_Z^{in}(u) \geq k + n - 1$. If $k_{max}(u, G) \geq k + 1$, 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 G to Z', u does not lose any in-neighbor with k_{max} values no smaller than $k + 1$. Hence, in Z', $deg_{Z'}^{in}(u) \geq k + 1$. 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 2 and the first part of this proof, it is obvious that if $k_{max}(v, G) \neq k$, $k_{max}(v, G)$ cannot change. Combining all of the above, Theorem 9 is proved.

B. Detailed algorithms for edge group construction in Section V-A

The construction of edge groups is described in Algorithm 5. 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 8. 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 5 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 $k_{max}(u_i, G) = x$, 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).

C. Extra details for Algorithm 2 in Section IV-A

We provide more details of the $k_{max}(v, G)$ maintenance algorithm for single-edge insertions in Algorithm 2. The details are shown in Function REMOVE, which can be used at line 19 of Algorithm 2. We can see that deleted[v] is marked as true (line1). Then, for all the out-neighbors w of v, if $k_{max}(w, G)$ is x, deleted[w] will be decremented by 1(line 4). If deleted[w] is x after the decrement, a recursive deletion will be ignited from w.

D. Detailed algorithm for $l_{max}(v, k, G)$ maintenance in Section IV-A

We introduce the detailed algorithm for $l_{max}(v, k, G)$ maintenance in Section IV-A. With obtained $\{k_{max}(v, G') : \forall v \in V_{G'}\}$, how $l_{max}(k)$ are maintained is presented in Al-

Algorithm 6: 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), $\mathcal{K} = \{k_{max}(v, G') : \forall v \in V_{G'}\}$, 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 $(i,0)$-core in G;
4 Let G'_i be the updated $(i,0)$-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 $\Phi(G')$;

gorithm 6. 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, based on which Algorithm 6 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 ($0 \leq k \leq M-1$) is independent of each other, providing an opportunity for parallel execution. Specifically, for lines 5-6 of Algorithm 6, 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 in G', we omit the details for brevity.

E. Detailed overall H-index-based algorithm for batch insertion in Section V-C

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 7, 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 $\mathcal{K} = \{k_{max}(v, G) : \forall v \in V_G\}$ based on Strategy 1 and 2 (line 1). Next, for each generated group B, the algorithm maintains $\mathcal{K}$ using Algorithm 4, 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 algorithm inserts it into or deletes it from G_k and maintains $l_{max}(k)$ using an algorithm similar to Algorithm 4 (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.

Algorithm 7: H-index-based D-core maintenance algorithm with a batch insertion

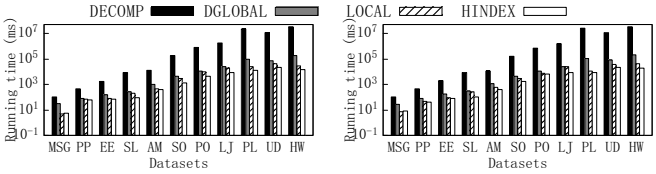
Input: digraph $G = (V_G, E_G)$, inserted edges E_i, original anchored corenesses $\Phi(G)$
Output: updated anchored corenesses $\Phi(G')$ for G' with $V_{G'} = V_G$ and $E_{G'} = E_G \cup E_i$

1 Construct edge groups $\mathcal{B}$ based on strategies in Section V-A 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 4(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 V-A;
16 **for** $A \in \mathcal{A}$ **do**
17 Insert/delete A into/from G_k;
18 Maintain $l_{max}(v,k,G)$ for each vertex $v \in V_{G'_k}$ with a $l_{max}(v,k,G)$ maintenance algorithm similar to the $k_{max}(v,G)$ maintenance algortihm in Algorithm 4;
19 **for** *edge* $e' \in E_D$ *but* $\notin \mathcal{A}$ **do**
20 Insert/delete e' into/from G_k;
21 Maintain $l_{max}(v,k,G)$ for each vertex $v \in V_{G'_k}$ similarly to Algorithm 2;
22 **return** the updated anchored corenesses as $\Phi(G')$;

F. Additional experiments

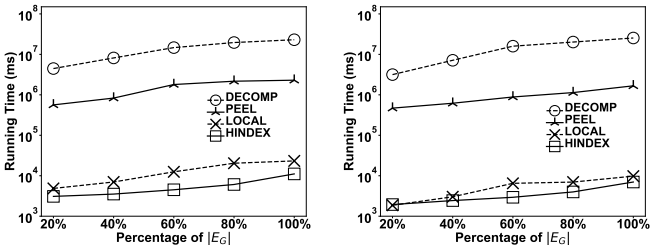
We provide additional experiments to test the effect of cardinality on synthetic graphs. An experiment on the D-core maintenance application in community search is also provided.

Exp-8: D-core maintenance application in community search. In this experiment, we show the D-core maintenance application in community search over all the datasets we

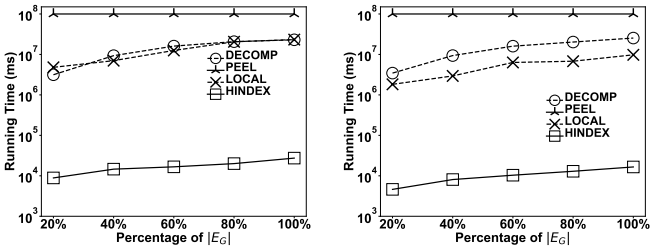


(a) Edge deletions (b) Edge insertions

Fig. 11. Case study of D-core-based community search



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



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

Fig. 12. Effect of cardinality over synthetic graphs

use. We compare three methods: i) removing disqualified vertices to obtain the D-core community in the updated graph [9] (denoted as DGLOBAL); ii) computing the D-core decomposition of the updated graph using DECOMP, and obtaining the community based on the D-core decomposition result; iii) using the proposed D-core maintenance algorithms LOCAL and HINDEX to update the D-core decomposition of the updated graph, and obtaining the community. For each dataset we use, we randomly select a vertex and query the maximal D-core containing it for each query. We generate 100 queries and report the average running time. Figure 11 shows the result. As expected, our proposed LOCAL and HINDEX significantly expedite the D-core-based community search with edge updates, demonstrating the effectiveness of the proposed algorithms.

Exp-9: Effect of dataset cardinality on synthetic graphs. As shown in Figure 12, the running time of all the algorithms increases with the dataset cardinality as expected. In addition, we can see that compared to the result reported in Figure 6, LOCAL demonstrated better performance compared to DECOMP in Figure 12. The reason is that LOCAL only traverses a local subgraph to accomplish D-core maintenance, while DECOMP always computes the new D-core decomposition from scratch with the original graph. The larger the original graph is, the more obvious the difference is. Hence, LOCAL is more efficient compared to DECOMP on PL.

G. Complete proof to the unboundedness of D-core maintenance

We give complete proof of the unboundedness of the D-core maintenance problem. Specifically, to accomplish D-core maintenance with a given updated edge (u_1, v_1), we first need to maintain $k_{max}(v, G)$ for $v \in V_G$. Then, we maintain $l_{max}(v, k, G)$ for each vertex $v \in V_{G'}$ with $k \in [0, M]$, where $M = min\{k_{max}(u_1, G'), k_{max}(v_1, G')\}$. If we denote the visited subgraph for the maintenance of $l_{max}(v, k_1, G)$ as AFF^{k_1}, then we have to visit $\sum_{i=0}^{M} AFF^i$ to accomplish the complete $l_{max}(v, k, G)$ maintenance for each vertex $v \in V_G$. Note that $M \leq k^G_{max}$, where $k^G_{max} = max\{k_{max}(v, G) \mid \forall v \in V_G\}$. Since k^G_{max} is bounded by $|E_G|^{0.5}$ [8], the time complexity of l_{max} maintenance in D-core maintenance can be represented as $O(f(||\sum_{i=0}^{|E_G|^{0.5}} AFF^i||_c))$ for some polynomial function f and some positive integer c, demonstrating the unboundedness of the D-core maintenance problem.