Local Algorithms for Distance-generalized Core Decomposition over Large Dynamic Graphs

Qing Liu, Xuliang Zhu, Xin Huang, Jianliang Xu
Hong Kong Baptist University, Hong Kong, China
{qingliu,csxlzhu,xinhuang,xujl}@comp.hkbu.edu.hk

ABSTRACT

The distance-generalized core, also called (k, h)-core, is defined as the maximal subgraph in which every vertex has at least k vertices at distance no longer than h. Compared with k-core, (k, h)-core can identify more fine-grained subgraphs and, hence, is more useful for the applications such as network analysis and graph coloring. The state-of-the-art algorithms for (k, h)-core decomposition are peeling algorithms, which iteratively delete the vertex with the minimum h-degree (i.e., the least number of neighbors within h hops). However, they suffer from some limitations, such as low parallelism and incapability of supporting dynamic graphs. To address these limitations, in this paper, we revisit the problem of (k, h)-core decomposition. First, we introduce two novel concepts of *pairwise h-attainability index* and *n-order H-index* based on an insightful observation. Then, we thoroughly analyze the properties of n-order H-index and propose a parallelizable local algorithm for (k, h)-core decomposition. Moreover, several optimizations are presented to accelerate the local algorithm. Furthermore, we extend the proposed local algorithm to address the (k, h)-core maintenance problem for dynamic graphs. Experimental studies on real-world graphs show that, compared to the best existing solution, our proposed algorithms can reduce the (k, h)-core decomposition time by 1-3 orders of magnitude and save the maintenance cost by 1-2 orders of magnitude.

PVLDB Reference Format:
Qing Liu, Xuliang Zhu, Xin Huang, Jianliang Xu. Local Algorithms for Distance-generalized Core Decomposition over Large Dynamic Graphs. PVLDB, 14(9): 1531 - 1543, 2021.
doi:10.14778/3461535.3461542

1 INTRODUCTION

The identification of cohesive subgraphs, in which vertices form strong, intense, or frequent ties, is one of the major tasks for network analysis [47]. To this end, many models have been proposed, such as k-core [33], k-truss [26, 31, 45], maximal clique [12, 13], quasi-cliques [43], and k-plexes [7, 52]. Among them, the k-core model has received wide attention since its computation takes linear time and it is the basis of many other models. Specifically, a k-core is a subgraph in which every vertex's degree is no less than k. Correspondingly, the core decomposition for a graph aims to compute the coreness for each vertex (i.e., the maximum k such that the vertex is in a k-core).

Recently, Bonchi et al. [9] proposed a new model called distance-generalized core, also known as (k, h)-core. In a (k, h)-core, every vertex v has at least k vertices whose distance to v is at most h. Generalized from k-core,[1] the (k, h)-core model has several advantages in network analysis by considering more structural information of h-hop neighborhoods. First, the (k, h)-core model can find more fine-grained subgraphs. Take the graph in Figure 1 as an example. If we employ the k-core model, the whole graph is identified as a 2-core. In contrast, if we set $h = 2$, the (k, h)-core model can find three different cores: a $(4, 2)$-core (i.e., the whole graph), a $(5, 2)$-core (i.e., the subgraph induced by the gray and white vertices), and a $(7, 2)$-core (i.e., the subgraph induced by the white vertices). By identifying more fine-grained subgraphs, the (k, h)-core model can help us better understand the hierarchy structure of the graph. Second, by varying h, the (k, h)-core model can discover dense subgraphs of different structures. For instance, Figure 2 shows a case study of two (k, h)-cores on a web graph *web-polblogs*.[2] Clearly, the two cores exhibit very different structures: the nodes of the $(k, 1)$-core in Figure 2(a) has a *peer* relationship since they densely connect to each other directly; the $(k, 2)$-core in Figure 2(b) has a *leader-follower* structure, where the white (follower) nodes connect to the black (leader) node directly but the white nodes do not densely connect to each other.

Besides network analysis, the (k, h)-core model has many other applications, such as distance-h graph coloring, maximum h-club computation, landmarks selection, and community search [9]. For example, in distance-h graph coloring, which can be applied to various tasks such as schedule making, register allocation, and seating planning, the distance-h chromatic number is defined as the minimum number of colors needed to color a graph such that any two vertices having the same color are more than h hops apart [35]. While finding the exact distance-h chromatic number is NP-hard, an upper bound can be obtained by efficiently computing the maximum coreness of non-empty (k, h)-cores [9].

In [9], Bonchi et al. devised a set of algorithms for (k, h)-core decomposition, i.e., computing the coreness of each vertex given a distance threshold h. However, these algorithms suffer from two major limitations.

- First, the algorithms have low parallelism. Specifically, the algorithms proposed in [9] are peeling algorithms, which iteratively delete the vertex with the minimum h-degree (i.e., the least number of neighbors within h hops). But the peeling algorithms have two bottlenecks: (1) it needs global graph information to find the minimum h-degree vertex for deletion,

Proceedings of the VLDB Endowment, Vol. 14, No. 9 ISSN 2150-8097.
doi:10.14778/3461535.3461542

[1] A k-core is equivalent to a (k, h)-core with $h = 1$.
[2] http://networkrepository.com/web-polblogs.php

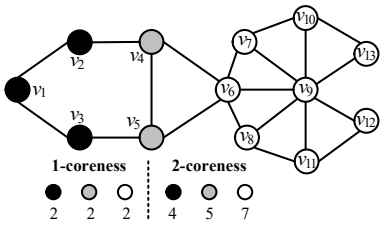
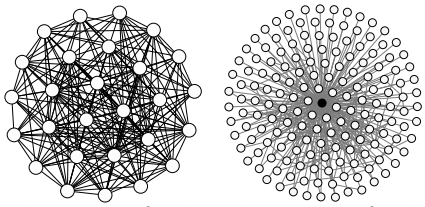


Figure 1: (k, h)-core



(a) Case study I ($h = 1$) (b) Case study II ($h = 2$)

Figure 2: Case Study for (k, h)-core

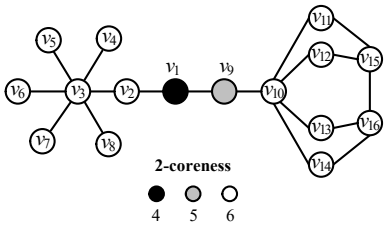


Figure 3: (k, h)-core ($h = 2$)

and (2) it entails sequential processing, i.e., the vertices are processed from low h-degree to high h-degree. These lead to low parallelism of the peeling algorithms, making them unable to handle large-scale graphs.

- Second, the proposed algorithms cannot support dynamic graphs efficiently. In real-world applications, some graphs are highly dynamic. The structures of these graphs may frequently change by insertion/deletion of nodes/edges over time. For example, in a web graph, the update of web contents may result in creation of new links or removal of existing links. For dynamic graphs, many applications, such as interactive graph visualization [44], require maintaining the corenesses in real time. Although one can reevaluate the corenesses upon each graph update by running a (k, h)-core decomposition algorithm from scratch, it is inefficient especially when graph updates are frequent. More efficient algorithms are desired to maintain the (k, h)-core over a dynamic graph.

Motivated by this, we revisit the problem of (k, h)-core decomposition in this paper. Inspired by [32], we find that we do not need global graph information to perform the (k, h)-core decomposition. More specifically, given a graph G and a positive integer h, a vertex v's coreness is i iff: (1) there exists a vertex set V_v in G such that (i) $|V_v| = i$, (ii) $\forall v' \in V_v$, the coreness of v' is no less than i, and (iii) $\forall v' \in V_v$, there exists a path p, whose length is no longer than h, from v' to v and the coreness of every vertex on p is no less than i; (2) there does not exist a vertex set V'_v in G such that (i) $|V'_v| = i + 1$, (ii) $\forall v' \in V'_v$, the coreness of v' is no less than $i + 1$, and (iii) $\forall v' \in V'_v$, there exists a path p, whose length is no longer than h, from v' to v and the coreness of every vertex on p is no less than $i + 1$. Take the graph in Figure 3 as an example. Assume $h = 2$. For vertex v_3, its coreness is 6, as we can find a qualified vertex set $V_{v_3} = \{v_2, v_4, v_5, v_6, v_7, v_8\}$, in which every vertex's coreness is 6. For vertex v_9, there exist six vertices, i.e., $v_2, v_{10}, v_{11}, v_{12}, v_{13}$, and v_{14}, whose coreness is 6 and can reach v_9 within 2 hops. But, for v_2, there does not exist a qualified path from v_2 to v_9. Hence, the qualified vertex set for v_9 is $V_{v_9} = \{v_{10}, v_{11}, v_{12}, v_{13}, v_{14}\}$, meaning that the coreness of v_9 is 5 rather than 6. In general, for any vertex v, we can just make use of the vertices, whose distance to v is no longer than the given h, to compute its coreness.

Following the above observation, we propose a new *local* algorithm for (k, h)-core decomposition. As our algorithm is oriented on vertices, which are independent of each other, it is highly parallelizable. Moreover, we present a series of optimizations including asynchronous processing, avoiding redundant computations, and employing upper bounds, to further enhance the performance of the local algorithm. In addition, we extend the proposed local algorithm to address the (k, h)-core maintenance problem for dynamic graphs. Specifically, we first filter out the vertices whose coreness does not need to be updated. Next, we employ the local algorithm to update the remaining vertices, during which we use each vertex' original coreness to accelerate the computation of its new coreness.

Overall, this paper's contributions are summarized as follows:

- We make an important observation for (k, h)-core decomposition, based on which we propose two novel concepts of pairwise h-attainability index and n-order H-index.
- We conduct a theoretical analysis on the n-order H-index, and develop a parallelizable local algorithm for (k, h)-core decomposition. Moreover, several optimizations are proposed to further improve the algorithm's performance.
- We explore the problem of (k, h)-core maintenance for dynamic graphs and extend the local algorithm to efficiently update the (k, h)-core.
- We conduct extensive experiments on real-world graphs to validate the efficiency of our proposed algorithms.

Roadmap: Section 2 defines the problem of (k, h)-core decomposition. Section 3 presents our theoretical basis, based on which Section 4 proposes the local algorithm. Section 5 extends the local algorithm to address the (k, h)-core maintenance problem. Section 6 presents our experimental results. Finally, we review the related work and conclude this paper in Sections 7 and 8, respectively.

2 PROBLEM FORMULATION

Let $G(V_G, E_G)$ be an undirected, simple, and unweighted graph with a set V_G of vertices and a set E_G of edges. A subgraph $H(V_H, E_H)$ of G satisfies $V_H \subseteq V_G$ and $E_H = \{(u, v) \in E_G : u, v \in V_H\}$. For a vertex v, we denote the neighbors of v in G as $\mathsf{N}_G(v) = \{u \in V(G) : (u, v) \in E_G\}$. We extend the concept of neighbors from 1-hop neighborhood to h-hop neighborhood. The set of h-neighbors of v in graph G is denoted by $\mathsf{N}_G(v, h) = \{u : u \in V_G \wedge \mathrm{dist}_G(v, u) \leq h\}$, where $\mathrm{dist}_G(v, u)$ is the shortest-path distance between v and u in G. Thus, $\mathsf{N}_G(v) = \mathsf{N}_G(v, 1)$. The h-degree of vertex v in G is defined as the cardinality of v's h-neighbors, i.e., $\deg_G(v, h) = |\mathsf{N}_G(v, h)|$. In addition, we denote $G_h[v]$ as a subgraph of G induced by the vertices $N_G(v, h)$. Without loss of generality, we assume that $h > 1$ holds throughout this paper, as in [9].

Definition 2.1. **((k, h)-core [9]).** *Given a graph G, two positive integers k and h, the (k, h)-core of G is a maximal subgraph $H \subseteq G$ satisfying $\forall v \in V_H$, $\deg_H(v, h) \geq k$.*

Based on (k, h)-core, we define h-coreness as follows.

Definition 2.2. **(h-coreness).** *Given a positive integer h, for a vertex $v \in V(G)$, the h-coreness of v is the largest number k such that there exists a (k,h)-*core *containing v, denoted by*

$$\mathrm{C}_G(v,h) = \arg \max_{\exists (k,h)\text{-core } H \subseteq G, v \in V_H} k.$$

Take the graph in Figure 1 as an example and assume that $h = 2$. The subgraph induced by the gray and white vertices is a $(5,2)$-core since every vertex's 2-degree in the subgraph is no less than 5. Vertex v_4 is in $(5,2)$-core but not in $(7,2)$-core. Hence, the 2-coreness of v_4 is 5. For simplicity, hereafter we simply use coreness to refer to h-coreness and use $\mathrm{C}(v,h)$ to denote $\mathrm{C}_G(v,h)$. Next, we present an important concept of (k,h)-*attainability*, which is a natural property of (k,h)-core.

Definition 2.3. **((k,h)-*attainability*).** *Given two vertices v and u in G, we say v and u are (k,h)-attainable if and only if there exists a path $p = (v_1, ..., v_l)$ in (k,h)-*core *$H \subseteq G$ such that (i) $v_1 = v$, $v_l = u$; (ii) the length l of path p satisfies $l \leq h$; and (iii) $\min_{1 \leq i \leq l}(\mathrm{C}_G(v_i,h)) \geq k$.*

The property of (k,h)-attainability is very useful for our algorithm development, which gives a new view of (k,h)-core analysis in terms of path connectivity w.r.t. the coreness and distance constraints. For example, in Figure 1, v_4 and v_9 are $(5,2)$-attainable since in the path $p = (v_4, v_6, v_9)$, $\mathrm{C}_G(v_6,2) = \mathrm{C}_G(v_9,2) = 7$ and $\mathrm{C}_G(v_4,2) = 5$. In addition, the (k,h)-core has two more structural properties, i.e., (i) *uniqueness*: for two given values of k and h, the (k,h)-core is unique; and (ii) *containment*: if h is fixed, the $(k+1,h)$-core is a subgraph of the (k,h)-core. Based on the above concepts, we formally define the problem studied in this paper.

Problem. **((k,h)-core Decomposition in Dynamic Graphs).** *Given a graph G and a positive integer h, the (k,h)-core decomposition is to compute the coreness for every vertex in G. Moreover, when graph G undergoes the updates of edge insertions/deletions, it needs to update all vertices' corenesses accordingly.*

It is worth mentioning that the problem definition includes two aspects: (1) computing the coreness for all vertices, which assumes that the graph is static; and (2) updating the coreness for all vertices when the graph changes, which is also called (k,h)-core maintenance. Correspondingly, we propose efficient algorithms for these two aspects in Sections 4 and 5, respectively. When $h = 1$, the (k,h)-core decomposition corresponds to the well-known problem of core decomposition, which has been extensively studied in the literature (see Section 7 for a detailed survey).

3 THEORETICAL BASIS

In this section, we provide a detailed theoretical analysis, which lays the foundation for our algorithms. Specifically, we first introduce two novel concepts of *pairwise h-attainability index* and *n-order H-index*, and then analyze the properties of n-order H-index.

3.1 n-Order H-Index

Motivations based on H-index. Recall that, according to our observation in Section 1, if a vertex v's coreness is k, a part of conditions are: (1) there exists a vertex set V_v in G such that (i) $|V_v| = k$ and (ii) $\forall v' \in V_v$, the coreness of v' is no less than k. (2) there does not exist a vertex set V'_v in G such that (i) $|V'_v| = k+1$, (ii) $\forall v' \in V'_v$, the coreness of v' is no less than $k+1$. It is very similar to the semantic of H-index, which is first proposed by Jorge E. Hirsch [23] to measure the citation impact of a scholar or a journal. For a scholar, the H-index is defined as the maximum value of l such that there exist at least l papers whose citation is no smaller than l. Formally, given a set of integers $S = \{x_1, x_2, ..., x_n\}$, we denote the operator $\mathcal{H}(\cdot)$ to compute the H-index, i.e., $\mathcal{H}(S) = \mathcal{H}(\{x_1, x_2, ..., x_n\}) = \arg\max_k(|\{x \in S : x \geq k\}| \geq k)$. For example, $S = \{1,2,3,4,5,6,7\}$, the H-index of S is $\mathcal{H}(S) = 4$ as there exist 4 integers $\{4,5,6,7\}$ that are no less than 4.

However, the direct application of H-index cannot help resolve (k,h)-core decomposition as the current H-index definition does not capture the path constraint of h-hop reachability, which is shown as (k,h)-*attainability* in Definition 2.3. As such, we propose a novel (k,h)-core based H-index (to be formally defined in Definition 3.2) for the computation of corenesses. To start with, we first introduce a concept of *pairwise h-attainability index*.

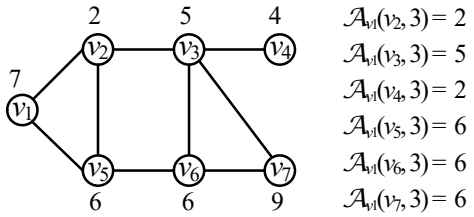


Figure 4: Example of Pairwise h-attainability Index (h=3)

Pairwise h-attainability Index. Assuming that each vertex $v \in V(G)$ is associated with an H-index, denoted as $\mathrm{H}(v)$, we give a definition of pairwise h-attainability index as follows.

Definition 3.1. **(Pairwise h-attainability Index).** *Given two vertices u, v in G and a positive integer h, the pairwise h-attainability index of u w.r.t. v, denoted by $\mathcal{A}_v(u,h)$, is*

$$\mathcal{A}_v(u,h) = \max_{\forall p \in P}\Big(\min_{\forall w(\neq v) \in p} \mathrm{H}(w)\Big) \quad (1)$$

where P is the set of paths from u to v whose distance is no longer than h.

In a word, the pairwise h-attainability index of u w.r.t. v is the maximum of the minimum H-indexes for every path from u to v whose distance is no longer than h. An example is given in Figure 4, where each vertex's H-index is shown by the number next to it. We list the vertices' pairwise 3-attainability indexes w.r.t. v_1 on the right part of the figure. For v_3, there exist three paths (no longer than 3 hops) from v_3 to v_1, i.e., $p_1 = (v_3, v_2, v_1)$, $p_2 = (v_3, v_2, v_5, v_1)$, and $p_3 = (v_3, v_6, v_5, v_1)$. The minimum H-indexes for p_1, p_2, and p_3 are 2, 2, and 5, respectively. Hence, the pairwise 3-attainability index of v_3 w.r.t. v_1 is 5. Note that, without the need of enumerating all possible paths, the pairwise h-attainability index can be more efficiently computed using an incremental method, as will be shown in Section 4.1. For conciseness, we simply call the pairwise h-attainability index the *attainability index* in the rest of the paper.

***n*-order H-index**. Based on the attainability index introduced above, we formally define the n-order H-index.

Definition 3.2. **(n-order H-index).** *Given a graph G, a positive integer h, and a vertex $v \in V_G$, the n-order H-index of vertex v w.r.t. h, denoted by $\mathsf{H}_G^{(n)}(v,h)$, is defined as*

$$\mathsf{H}_G^{(n)}(v,h) = \begin{cases} \deg_G(v,h) & n = 0 \\ \mathcal{H}(\mathcal{A}_v^{(n-1)}(u_1,h), ..., \mathcal{A}_v^{(n-1)}(u_i,h)) & n > 0 \end{cases} \quad (2)$$

where $i = \deg_G(v,h)$ and $\forall j \in [1,i]$, $u_j \in \mathsf{N}_G(v,h)$; $\mathcal{A}_v^{(n)}(u,h)$ denotes the vertex u's n-order attainability index w.r.t. v and h, which is computed in the same way as in Equation 1. Specifically,

$$\mathcal{A}_v^{(n)}(u,h) = \max_{\forall p \in P}(\min_{\forall w(\neq v) \in p} \mathsf{H}_G^{(n)}(w,h)) \quad (3)$$

By Definition 3.2, a vertex v's 0-order H-index equals v's h-degree, i.e., $\mathsf{H}_G^{(0)}(v,h) = \deg_G(v,h)$; for $n \geq 1$, v's n-order H-index is the H-index value of all v's h-neighbors' $(n-1)$-order attainability indexes, i.e., $\mathsf{H}_G^{(n)}(v,h) = \mathcal{H}(S^{(n-1)})$ where $S^{(n-1)} = \{\mathcal{A}_v^{(n-1)}(u,h) : u \in \mathsf{N}_G(v,h)\}$. Take vertex v_1 in Figure 4 as an example. Assume that $h = 2$. $\mathsf{H}_G^{(0)}(v_1,2) = \deg_G(v_1,2) = 4$. $\mathsf{H}_G^{(1)}(v_1,2) = \mathcal{H}(\mathcal{A}_{v_1}^{(0)}(v_2,2), \mathcal{A}_{v_1}^{(0)}(v_3,2), \mathcal{A}_{v_1}^{(0)}(v_5,2), \mathcal{A}_{v_1}^{(0)}(v_6,2)) = \mathcal{H}(6,6,5,5) = 4$. Note that when the context is clear, we drop subscripts and use $\mathsf{H}^{(n)}(v,h)$ instead of $\mathsf{H}_G^{(n)}(v,h)$.

3.2 Convergence and Asynchrony

In this section, we analyze the properties of n-order H-index $\mathsf{H}_G^{(n)}(v,h)$, including the convergence, convergence bound, asynchrony, and relative independence of 0-order H-index. All these properties are critically important for developing correct and fast algorithms. Due to space limitations, we omit the proof for the lemmas and theorems that intuitively hold.

3.2.1 Convergence.

Theorem 3.1. ***(Monotonicity)*** *Given a vertex v in graph G, $\forall n \in \mathbb{N}$, it holds that $\mathsf{H}^{(n)}(v,h) \geq \mathsf{H}^{(n+1)}(v,h)$.*

Proof. We prove the theorem by mathematical induction.

(i) For $n = 0$, $\forall v \in V_G$, $\mathsf{H}^{(0)}(v,h) = \deg_G(v,h)$. $\mathsf{H}^{(1)}(v,h) = \mathcal{H}(\mathcal{A}_v^{(0)}(u_1,h), ..., \mathcal{A}_v^{(0)}(u_i,h))$ where $i = \deg_G(v,h)$. Thus, $\mathsf{H}^{(1)}(v,h) \leq \deg_G(v,h) = \mathsf{H}^{(0)}(v,h)$.

(ii) Assume that the theorem holds for $n = m$, i.e., $\mathsf{H}^{(m)}(v,h) \geq \mathsf{H}^{(m+1)}(v,h)$. We prove $\mathsf{H}^{(m+1)}(v,h) \geq \mathsf{H}^{(m+2)}(v,h)$ as follows.

Based on Equation 3, $\mathcal{A}_v^{(m)}(u,h) \geq \mathcal{A}_v^{(m+1)}(u,h)$. We have

$$\begin{aligned} \mathsf{H}^{(m+2)}(v,h) &= \mathcal{H}(\mathcal{A}_v^{(m+1)}(u_1,h), ..., \mathcal{A}_v^{(m+1)}(u_i,h)) \\ &\leq \mathcal{H}(\mathcal{A}_v^{(m)}(u_1,h), ..., \mathcal{A}_v^{(m)}(u_i,h)) \\ &= \mathsf{H}^{(m+1)}(v,h) \end{aligned}$$

Hence, $\mathsf{H}^{(m+1)}(v,h) \geq \mathsf{H}^{(m+2)}(v,h)$ holds.

Based on (i) and (ii), the theorem is proved. □

According to Theorem 3.1, the n-order H-index $\mathsf{H}^{(n)}(v,h)$ is *monotonic* and *non-increasing* w.r.t. the increasing n. Based on the definitions of H-index operator $\mathcal{H}(\cdot)$ and attainability index $\mathcal{A}(\cdot)$, $\mathsf{H}^{(n)}(v,h)$ is always *a non-negative integer*. Hence, when the number n is large enough, $\mathsf{H}^{(n)}(v,h)$ can converge to *a nonnegative value*.

Lemma 3.1. *For two vertices $v \neq u$ in G and $\forall n \in \mathbb{N}$, it holds that $\mathcal{A}_u^{(n)}(v,h) \geq \min_{w \in V_G} \mathsf{H}^{(n)}(w,h)$.*

Proof. $\mathcal{A}_v^{(n)}(u,h) = \max_{\forall p \in P}(\min_{\forall w(\neq v) \in p} \mathsf{H}_G^{(n)}(w,h)) \geq \min_{\forall p \in P}(\min_{\forall w(\neq v) \in p} \mathsf{H}_G^{(n)}(w,h)) \geq \min_{w \in V_G} \mathsf{H}^{(n)}(w,h)$. □

Lemma 3.2. *Given a graph G and a positive integer h, let $\deg_{min}(G,h) = \min_{w \in V_G} \deg_G(w,h)$. For $\forall v \in V_G$ and $\forall n \in \mathbb{N}$, we have $\mathsf{H}^{(n)}(v,h) \geq \deg_{min}(G,h)$.*

Proof. We prove the theorem by mathematical induction.

(i) For $n = 0$, $\mathsf{H}^{(0)}(v,h) = \deg_G(v,h) \geq \deg_{min}(G,h)$ holds.

(ii) Assume that the theorem holds for $n = m$, i.e., $\forall v \in V_G$, $\mathsf{H}^{(m)}(v,h) \geq \deg_{min}(G,h)$. By Lemma 3.1, $\mathcal{A}_u^{(m)}(v,h) \geq \min_{w \in V_G} \mathsf{H}^{(m)}(w,h) \geq \deg_{min}(G,h)$. For $\mathsf{H}^{(m+1)}(v,h) = \mathcal{H}(\mathcal{A}_v^{(m)}(u_1,h), ..., \mathcal{A}_v^{(m)}(u_i,h))$, there exist at least $\deg_{min}(G,h)$ cases for different u_j's such that $\mathcal{A}_v^{(m)}(u_j,h) \geq \deg_{min}(G,h)$. Thus, $\mathsf{H}^{(m+1)}(v,h) \geq \deg_{min}(G,h)$.

Based on (i) and (ii), the theorem is proved. □

Lemma 3.3. *Given a graph G and a subgraph $G' \subseteq G$, $\forall v \in V_{G'}$ and $\forall n \in \mathbb{N}$, it holds that $\mathsf{H}_G^{(n)}(v,h) \geq \mathsf{H}_{G'}^{(n)}(v,h)$.*

Based on the above lemmas and Theorem 3.1, we show the limitation of $\mathsf{H}^{(n)}(v,h)$.

Theorem 3.2. ***(Convergence)***

$$\lim_{n \to \infty} \mathsf{H}^{(n)}(v,h) = \mathsf{C}(v,h) \quad (4)$$

Proof. We prove the theorem from the following two aspects:

(i) $\mathsf{H}^{(\infty)}(v,h) \geq \mathsf{C}(v,h)$. Let $G' \subseteq G$ be a $(\mathsf{C}(v,h),h)$-core that contains v. Then, we have $\deg_{min}(G',h) = \mathsf{C}(v,h)$. Combining Lemmas 3.3 and 3.2, $\forall n \in \mathbb{N}$, $\mathsf{H}_G^{(n)}(v,h) \geq \mathsf{H}_{G'}^{(n)}(v,h) \geq \deg_{min}(G',h) = \mathsf{C}(v,h)$. Hence, $\mathsf{H}^{(\infty)}(v,h) \geq \mathsf{C}(v,h)$.

(ii) $\mathsf{H}^{(\infty)}(v,h) \leq \mathsf{C}(v,h)$. Let the vertex set $U = \{u : u \in V_G \text{ and } \mathsf{H}^{(\infty)}(u,h) \geq \mathsf{H}^{(\infty)}(v,h)\}$ and $G'' \subseteq G$ be the subgraph of G induced by $U \cup v$. For the vertex v, according to Definition 3.2, $\mathsf{H}^{(\infty)}(v,h) = \mathcal{H}(\mathcal{A}_v^{(\infty)}(u_1,h), ..., \mathcal{A}_v^{(\infty)}(u_i,h))$. We can find a vertex set U' from v's h-neighbors such that (i) $|U'| = \mathsf{H}^{(\infty)}(v,h)$, and (ii) $\forall u \in U'$, $\mathcal{A}_v^{(\infty)}(u,h) \geq \mathsf{H}^{(\infty)}(v,h)$. Based on Equation 3, $\mathsf{H}^{(\infty)}(u,h) \geq \mathcal{A}_v^{(\infty)}(u,h) \geq \mathsf{H}^{(\infty)}(v,h)$. Hence, $\forall u \in U'$, $\mathsf{H}^{(\infty)}(u,h) \geq \mathsf{H}^{(\infty)}(v,h)$, meaning that $U' \subseteq U$. Therefore, $\deg_{G''}(v,h) \geq |U'| = \mathsf{H}^{(\infty)}(v,h)$. In the same way, we can prove that $\forall u \in U, \deg_{G''}(u,h) \geq \mathsf{H}^{(\infty)}(v,h)$. Thus, G'' is a $(\mathsf{H}^{(\infty)}(v,h), h)$-core. If vertex v has been in a $(\mathsf{H}^{(\infty)}(v,h), h)$-core, its coreness is at least $\mathsf{H}^{(\infty)}(v,h)$, i.e., $\mathsf{C}(v,h) \geq \mathsf{H}^{(\infty)}(v,h)$.

Combining (i) and (ii), the theorem is proved. □

Theorem 3.2 implies that the vertex's n-order H-index finally converges to its coreness. Take vertex v_2 in Figure 4 as an example. We can find that, $\mathsf{H}^{(0)}(v_2,2) = 6$, $\mathsf{H}^{(1)}(v_2,2) = \mathsf{H}^{(2)}(v_2,2) = 4 = \mathsf{C}(v_2,2)$. It is worth mentioning that Theorem 3.2 lays the correctness foundation of the local algorithm to be proposed in Section 4.1.

3.2.2 Convergence Bound. Now, we show that $\mathsf{H}^{(n)}(v,h)$ can converge to $\mathsf{C}(v,h)$ in a limited number of steps, which is denoted as *convergence bound*. It guarantees that our proposed local algorithm can run efficiently. First, we introduce a concept of *h-degree hierarchy*.

Definition 3.3. **(*h*-degree Hierarchy).** *Given a graph G and a positive integer h, the i-th ($i \in \mathbb{N}$) h-degree hierarchy, denoted by D_i, is the set of vertices that have the minimum h-degree in G' where G' is a subgraph of G induced by the vertex set $V_G \setminus \bigcup_{0\le j<i} \mathsf{D}_j$. Formally,*

$$\mathsf{D}_i = \{v : \underset{v\in V_G\setminus\bigcup_{0\le j<i}\mathsf{D}_j}{\arg\min} \deg_{G'}(v,h)\} \tag{5}$$

Consider the graph G in Figure 4 and $h = 2$. Since both v_1 and v_4 have the minimum 2-degree, which is equal to 4, $\mathsf{D}_0 = \{v_1, v_4\}$. After deleting v_1 and v_4 from the graph, the vertices in the remaining subgraph have the same 2-degree, which is 4. Thus, $\mathsf{D}_1 = \{v_2, v_3, v_5, v_6, v_7\}$. For the vertices belonging to different h-degree hierarchies, we have the following theorem.

Lemma 3.4. *Given two vertices $v_i \in \mathsf{D}_i$, $v_j \in \mathsf{D}_j$, if $i \le j$, it holds that $\mathsf{C}(v_i,h) \le \mathsf{C}(v_j,h)$.*

Theorem 3.3. ***(Convergence Bound)** Given a graph G and a vertex $v \in \mathsf{D}_i$, then for $n \ge i$, $\mathsf{H}^{(n)}(v,h) = \mathsf{C}(v,h)$.*

Proof. We prove the theorem by induction on i.

When $i = 0$, D_0 is the set of vertices that have the minimum h-degree in the graph G. It is obvious that G is the maximal (k,h)-core for vertices of D_0. Thus, $\forall v \in \mathsf{D}_0$, $\mathsf{H}^{(0)}(v,h) = \deg_G(v,h) = \mathsf{C}(v,h)$.

Assume that the theorem is valid up to $i = m$, i.e., $\forall u \in \mathsf{D}_j (j \le m)$, for $n \ge j$, $\mathsf{H}^{(n)}(u,h) = \mathsf{C}(u,h)$. Let $v \in \mathsf{D}_{m+1}$. For the h-neighbors of v, we divide them into two sets S' and S''. Specifically, $S' = \{v' : v' \in \mathsf{N}_G(v,h) \wedge \exists j < m+1, v' \in \mathsf{D}_j\}$ and $S'' = \{v'' : v'' \in \mathsf{N}_G(v,h) \wedge \exists j \ge m+1, v'' \in \mathsf{D}_j\}$. The $(m+1)$-order H-index of v is: $\mathsf{H}^{(m+1)}(v,h) = \mathcal{H}(\mathcal{A}_v^{(m)}(v'_1,h), \mathcal{A}_v^{(m)}(v'_2,h), ..., \mathcal{A}_v^{(m)}(v'_x,h), ..., \mathcal{A}_v^{(m)}(v''_1,h), \mathcal{A}_v^{(m)}(v''_2,h), ..., \mathcal{A}_v^{(m)}(v''_y,h), ...)$, where $v'_x \in S'$ and $v''_y \in S''$. (i) For $\forall v'_x \in S'$, according to the above assumption, Lemma 3.4 and Equation 3, $\mathcal{A}_v^{(m)}(v'_x,h) \le \mathsf{H}^{(m)}(v'_x,h) = \mathsf{C}(v'_x,h) \le \mathsf{C}(v,h)$. (ii) Let G' be the subgraph of G induced by the vertex set $\bigcup_{j\ge m+1}\mathsf{D}_j$. Obviously, G' is a $(\deg_{G'}(v,h),h)$-core. Thus, $\mathsf{C}(v,h) \ge \deg_{G'}(v,h) = |S''|$. Combining (i) and (ii), $\mathsf{H}^{(m+1)}(v,h) \le \mathsf{C}(v,h)$. In addition, according to Theorems 3.1 and 3.2, $\mathsf{H}^{(m+1)}(v,h) \ge \mathsf{C}(v,h)$. Hence, $\mathsf{H}^{(m+1)}(v,h) = \mathsf{C}(v,h)$. Therefore, the theorem holds when $i = m+1$. □

Theorem 3.3 indicates that for each vertex $v \in D_i$, the n-order H-index of v converges to v's coreness at the step of $n = i$.

3.2.3 Asynchrony. Next, we introduce the property of asynchrony for n-order H-index, which can be used to optimize the local algorithms (Section 4.2). First, we introduce the concept of systematic time step t (we call it *round t* for short). At each round, every vertex can compute its n-order H-index *only once*. Assume that at the initial *round* t_0, we have computed the vertices' H-indexes in different orders. Specifically, we assume that for each $v_i \in V_G$, we have computed $\mathsf{H}^{(n_i)}(v_i,h)$, where n_i's for different v_i's could be different. Based on it, we denote $\mathsf{H}^t(v,h)$ as the n-order H-index of v computed at *round t* based on the n-order H-indexes of v's h-neighbors at *round* $t-1$. Formally,

$$\mathsf{H}^t(v,h) = \min(\mathsf{H}^{t-1}(v,h), \mathcal{H}(\mathcal{A}_v^{t-1}(u_1,h), ..., \mathcal{A}_v^{t-1}(u_i,h))) \tag{6}$$

where $\mathcal{A}_v^{t-1}(u_i,h)$ is the attainability index of v's h-neighbor u_i at *round* $t-1$, which is computed in the same way as in Equation 3. Then, we have the following theorem.

Theorem 3.4. ***(Asynchrony)** Given the n-order H-indexes of all vertices at start time, $\forall v \in V_G$, it holds that*

$$\lim_{t\to\infty} \mathsf{H}^t(v,h) = \mathsf{C}(v,h) \tag{7}$$

3.2.4 Relative Independence of 0-order H-index. Theorem 3.2 converges a vertex v's n-order H-index to its coreness, which starts from v's h-degree. The following theorem shows that v's coreness can also be converged from other values.

Theorem 3.5. ***(Relative Independence)** For each vertex $v \in V_G$, we set*

$$\mathsf{H}_G^{(n)}(v,h) = \begin{cases} r_v & n = 0 \\ \min(\mathsf{H}_G^{(n-1)}(v,h), \mathcal{H}(S^{(n-1)})) & n > 0 \end{cases} \tag{8}$$

where $r_v \ge \mathsf{C}(v,h)$ and $S^{(n-1)} = \{\mathcal{A}_v^{(n-1)}(u,h) : u \in \mathsf{N}_G(v,h)\}$, then it holds that

$$\lim_{n\to\infty} \mathsf{H}_G^{(n)}(v,h) = \mathsf{C}(v,h) \tag{9}$$

Thus, as long as $\mathsf{H}^0(v,h)$ is no less than its coreness $\mathsf{C}(v,h)$, the n-order H-index of v can finally converge to $\mathsf{C}(v,h)$. This property is greatly helpful to the optimization of our local algorithm (Section 4.2) and (k,h)-core maintenance (Section 5). It is worth mentioning that in Equations 6 and 8, the operation min() is used to ensure that the n-order H-index is *monotonic* and *non-increasing* w.r.t. the increasing n and finally converges to the coreness of v.

4 LOCAL ALGORITHM FOR (k,h)-CORE DECOMPOSITION

In this section, we first present a local algorithm for (k,h)-core decomposition based on the n-order H-index and its theoretical results in Section 3. Then, we propose several optimizations to accelerate the algorithm.

4.1 Local Algorithm

According to Theorem 3.2, the n-order H-index $\mathsf{H}^{(n)}(v,h)$ finally converges to its coreness when n is large enough. Thus, a basic method for (k,h)-core decomposition is to iteratively compute $\mathsf{H}^{(n)}(v,h)$ for all vertices. One important issue of this method is that we need to determine when to stop the iterative computation. We propose to terminate the iterations when $\mathsf{H}^{(n)}(v,h) = \mathsf{H}^{(n-1)}(v,h)$ for each vertex v. It is because if $\mathsf{H}^{(n)}(v,h) = \mathsf{H}^{(n-1)}(v,h)$, we have $\mathsf{H}^{(n+1)}(v,h) = \mathcal{H}(\mathcal{A}_v^{(n)}(u_1,h),..., \mathcal{A}_v^{(n)}(u_i,h)) = \mathcal{H}(\mathcal{A}_v^{(n-1)}(u_1,h), ..., \mathcal{A}_v^{(n-1)}(u_i,h)) = \mathsf{H}^{(n)}(v,h)$. Similarly, $\forall j > 1$, we can derive $\mathsf{H}^{(n+j)}(v,h) = \mathsf{H}^{(n-1)}(v,h)$, meaning $\mathsf{H}^{(n)}(v,h)$ has converged to its coreness.

Local algorithm for parallel (k,h)-core decomposition. We develop a local algorithm for (k,h)-core decomposition, which is outlined in Algorithm 1. Specifically, the algorithm first computes the h-degree for all vertices (lines 1-3). Then, it iteratively computes

Algorithm 1 Local Algorithm

Input: a graph G; a positive integer h;
Output: the h-coreness for each vertex v in G;
1: **for** each vertex $v \in V_G$ **in parallel do**
2: compute $\deg_G(v, h)$;
3: $\mathsf{H}^{(0)}[v] \leftarrow \deg_G(v, h)$;
4: $Tag \leftarrow$ **true**; $n \leftarrow 0$;
5: **while** $Tag =$ **true do**
6: $Tag \leftarrow$ **false**; $n \leftarrow n + 1$;
7: **for** each vertex $v \in V_G$ **in parallel do**
8: $\mathsf{H}^{(n)}[v] \leftarrow$ compute $\mathsf{H}^{(n)}(v, h)$ using Algorithm 2;
9: **if** $\mathsf{H}^{(n)}[v] \neq \mathsf{H}^{(n-1)}[v]$ **then**
10: $Tag \leftarrow$ **true**;
11: $\mathsf{C}(v, h) \leftarrow \mathsf{H}^{(n)}[v]$ for each vertex $v \in V_G$;
12: **return** $\{\mathsf{C}(v, h) : v \in V_G\}$;

$\mathsf{H}^{(i)}(v, h)$ using Algorithm 2 until the n-order H-index and $(n-1)$-order H-index are the same for all vertices (lines 5-10). Finally, it returns the h-coreness for all vertices (line 12). It is worth mentioning that the two procedures of Algorithm 1 are independent and can be efficiently implemented in parallel computing: (1) the computation of a vertex's h-degree is independent for different vertices (lines 1-3); and (2) at one particular round of iterations, the computation of $\mathsf{H}^{(i)}(v, h)$'s is independent of each other (lines 7-10).

Compute $\mathsf{H}^{(n)}(v, h)$. Algorithm 2 outlines the details of $\mathsf{H}^{(n)}(v, h)$ computation. It takes the $(n-1)$-order H-indexes of v's h-neighbors as inputs and outputs $\mathsf{H}^{(n)}(v, h)$. Specifically, the algorithm first computes the $(n-1)$-order attainability index for each vertex $u \in N_G(v, h)$ (lines 1-15), and then uses $\mathcal{H}(\cdot)$ operator to compute the H-index (line 16). It is worth noting that the $(n-1)$-order attainability index is computed using an incremental method by gradually expanding graph $G_h[v]$. It starts by computing the $(n-1)$-order attainability index of u whose distance to v is $d = 1$ (lines 2-4). Then, it iteratively checks each vertex u' that belongs to the v's h-neighbors, that can form a path to v within a distance of d, from $d = 2$ to $d = h$ (lines 6-15). If a vertex u' is newly visited or a longer path can generate a larger $(n-1)$-order attainability index, u''s $(n-1)$-order attainability index is assigned with the minimum value of its $(n-1)$-order H-index and the $(n-1)$-order attainability index of u''s predecessor (lines 11-13).

Example 1. *We use a graph G in Figure 5 to illustrate Algorithm 1. The detailed updating steps of n-order H-index is shown in Table 1. Specifically, $\mathsf{H}^{(0)}(\cdot)$ represents a vertex's h-degree. After three rounds of updating iterations, the n-order H-indexes of all vertices converge to their corenesses as $\mathsf{H}^{(3)}(\cdot) = \mathsf{H}^{(4)}(\cdot)$. Algorithm 1 takes a total of 4 iterations.*

Complexity Analysis. Let $\tilde{V}$ and $\tilde{E}$ be the vertex set and edge set of the maximum subgraph induced by a vertex's h-neighbors; r be the number of iterations to compute the n-order H-indexes. We analyze the complexity of Algorithm 1. Specifically, it runs $n = r$ iterations to compute $\mathsf{H}^{(n)}(v, h)$ for $v \in V(G)$. For computing $\mathsf{H}^{(n)}(v, h)$ in each iteration, Algorithm 2 needs to traverse the subgraph $G_h[v]$, which takes $O(h(|\tilde{V}| + |\tilde{E}|))$ time. As a result, the total time complexity of Algorithm 1 is $O(r|V_G|h(|\tilde{V}| + |\tilde{E}|))$. The time complexity of the state-of-the-art algorithm for (k, h)-core decomposition is $O(|V_G||\tilde{V}|(|\tilde{V}| + |\tilde{E}|))$ [9]. Both algorithms take $O(|\tilde{V}| + |\tilde{E}|)$ space. As will be verified in the experiments, r is very small. Thus, Algorithm 1 has a better efficiency performance.

Algorithm 2 $\mathsf{H}^{(n)}(v, h)$ Computation

Input: the set of $\{\mathsf{H}^{(n-1)}[u] : u \in N_G(v, h)\}$;
Output: the n-order H-index of v as $\mathsf{H}^{(n)}(v, h)$;
1: Initialization: $\mathcal{A}[.] \leftarrow 0$; $\mathcal{A}'[.] \leftarrow 0$; $visit[.] \leftarrow false$;
2: **for** each vertex $u \in \mathsf{N}_G(v, 1)$ **do**
3: $\mathcal{A}[u] \leftarrow \mathsf{H}[u]$; $\mathcal{A}'[u] \leftarrow \mathsf{H}[u]$
4: Add u into a queue Q;
5: $d \leftarrow 2$;
6: **while** $d \leq h$ **do**
7: Another queue: $Q' \leftarrow \varnothing$;
8: **while** $Q \neq \varnothing$ **do**
9: Pop a vertex u from Q; $visit[u] \leftarrow true$;
10: **for** each vertex $u' \in \mathsf{N}_{G_h[v]}(u, 1)$ **do**
11: **if** $visit[u'] =$ **false** or $\mathcal{A}'[u'] < \min(\mathsf{H}[u'], \mathcal{A}'[u])$ **then**
12: $\mathcal{A}[u'] \leftarrow \min(\mathsf{H}[u'], \mathcal{A}'[u])$;
13: Add u' into Q';
14: $d \leftarrow d + 1$; $Q \leftarrow Q'$;
15: $\mathcal{A}'[u] \leftarrow \mathcal{A}[u]$ for all $u \in N_G(v, h)$;
16: $\mathsf{H}^{(n)}(v, h) \leftarrow \mathcal{H}(\{\mathcal{A}[u] : u \in N_G(v, h)\})$ by Def. 3.2;
17: **return** $\mathsf{H}^{(n)}(v, h)$;

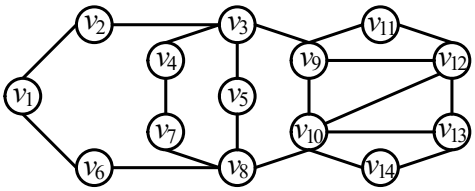


Figure 5: A Running Example

4.2 Optimizations

In this section, we propose three non-trivial optimizations to accelerate the local algorithm in Algorithm 1.

Optimization 1: Asynchronous processing.

In each iteration, Algorithm 1 computes $\mathsf{H}^{(n)}(v, h)$ based on the $(n-1)$-order H-index $\mathsf{H}^{(n-1)}(u, h)$ for $u \in N_G(v, h)$. However, before computing $\mathsf{H}^{(n)}(v, h)$, the $\mathsf{H}^{(n)}(u, h)$ values for some neighbors $u \in N_G(v, h)$ may have already been computed. Thus, if we use $\mathsf{H}^{(n)}(u, h)$ instead of $\mathsf{H}^{(n-1)}(u, h)$ to compute $\mathsf{H}^{(n)}(v, h)$, it helps the algorithm to converge faster. The correctness of this optimization is guaranteed by Theorem 3.4.

Example 2. *We illustrate Optimization 1 on graph G in Figure 5. The running steps are shown in Table 1. Compared with the synchronous processing, i.e., local algorithm in Table 1, the local algorithm integrated with Optimization 1 yields the same results using three iterations but converges faster.*

Optimization 2: Reducing redundant computations.

According to Theorem 3.3, for a vertex v that belongs to the h-degree hierarchy D_i, $\mathsf{H}^{(n)}(v, h)$ converges to its coreness within i iterations. Afterwards, $\mathsf{H}^{(n)}(v, h)$ does not change any more for $n \geq i$. However, Algorithm 1 continues to compute $\mathsf{H}^{(n)}(v, h)$ even if it has converged. This incurs unnecessary computations, which should be avoided. For example, for the local algorithm in Table 1, the vertex v_1 has converged to its coreness at $\mathsf{H}^{(0)}(v_1) = 4$. But, in the following four iterations, $\mathsf{H}^{(n)}(v_1, h)$ is still repeatedly

Table 1: Illustration of Local Algorithm and Three Optimizations on Graph G in Figure 5. Here, $h = 2$.

Method	$\mathsf{H}^{(n)}(\cdot)$	Vertex														
		v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}	v_{13}	v_{14}	
Local Algorithm	$\mathsf{H}^{(0)}(\cdot)$	4	6	10	6	8	6	6	11	10	10	5	7	6	5	$\Rightarrow 4\ Iterations$
	$\mathsf{H}^{(1)}(\cdot)$	4	4	6	6	6	4	6	6	6	6	5	5	5	5	
	$\mathsf{H}^{(2)}(\cdot)$	4	4	6	5	6	4	5	6	5	5	5	5	5	5	
	$\mathsf{H}^{(3)}(\cdot)$	4	4	5	5	5	4	5	5	5	5	5	5	5	5	
	$\mathsf{H}^{(4)}(\cdot)$	4	4	5	5	5	4	5	5	5	5	5	5	5	5	
Local Algorithm + Optimization 1	$\mathsf{H}^{(0)}(\cdot)$	4	6	10	6	8	6	6	11	10	10	5	7	6	5	$\Rightarrow 3\ Iterations$
	$\mathsf{H}^{(1)}(\cdot)$	4	4	6	5	6	4	5	6	6	6	5	5	5	5	
	$\mathsf{H}^{(2)}(\cdot)$	4	4	5	5	5	4	5	5	5	5	5	5	5	5	
	$\mathsf{H}^{(3)}(\cdot)$	4	4	5	5	5	4	5	5	5	5	5	5	5	5	
Local Algorithm + Optimization 2	$\mathsf{H}^{(1)}(\cdot)$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	$\Rightarrow 37.5\%\ Reduction$
	$\mathsf{H}^{(2)}(\cdot)$	**0**	1	1	1	1	1	1	1	1	1	**0**	1	1	**0**	
	$\mathsf{H}^{(3)}(\cdot)$	**0**	**0**	1	1	1	**0**	1	1	1	1	**0**	**0**	**0**	**0**	
	$\mathsf{H}^{(4)}(\cdot)$	**0**	**0**	1	**0**	1	**0**	**0**	1	**0**	**0**	**0**	**0**	**0**	**0**	
Local Algorithm + Optimization 3	$\mathsf{H}^{*(0)}(\cdot)$	4	6	10	6	8	6	6	11	10	10	5	7	6	5	$\Rightarrow 4\ Iterations$
	$\mathsf{H}^{*(1)}(\cdot)$	4	5	6	6	6	5	6	6	6	6	5	5	5	5	
	$\mathsf{H}^{*(2)}(\cdot)$	4	5	6	5	6	5	5	6	5	5	5	5	5	5	
	$\mathsf{H}^{*(3)}(\cdot)$	4	5	5	5	5	5	5	5	5	5	5	5	5	5	
	$\mathsf{H}^{*(4)}(\cdot)$	4	5	5	5	5	5	5	5	5	5	5	5	5	5	
	$\mathsf{H}^{(0)}(\cdot)$	4	5	5	5	5	5	5	5	5	5	5	5	5	5	$\Rightarrow 2\ Iterations$
	$\mathsf{H}^{(1)}(\cdot)$	4	4	5	5	5	4	5	5	5	5	5	5	5	5	
	$\mathsf{H}^{(2)}(\cdot)$	4	4	5	5	5	4	5	5	5	5	5	5	5	5	

computed. If we can identify such converged vertices as early as possible, it will improve the algorithm efficiency. On the other hand, the computation of h-degree hierarchy for a graph is expensive as mentioned previously. Thus, we propose the following lemma to avoid the unnecessary n-order H-index computation.

Lemma 4.1. *Given a graph G and a positive integer h, $\forall v \in V_G$ and $\forall n \in \mathbb{N}$, the following two cases hold:*

(1) *If $\forall u \in \mathsf{N}_G(v,h)$, $\mathsf{H}^{(n-2)}(u,h) = \mathsf{H}^{(n-1)}(u,h)$, it holds that $\mathsf{H}^{(n)}(v,h) = \mathsf{H}_G^{(n-1)}(v,h)$;*

(2) *Assume that $u \in \mathsf{N}_G(v,h)$ and $\mathsf{H}^{(n-2)}(u,h) > \mathsf{H}^{(n-1)}(u,h)$,*
   (a) *if $\mathsf{H}^{(n-2)}(u,h) \geq \mathsf{H}^{(n-1)}(v,h)$ and $\mathsf{H}^{(n-1)}(u,h) \geq \mathsf{H}^{(n-1)}(v,h)$, u will not trigger the decrease of $\mathsf{H}^{(n)}(v,h)$.*
   (b) *if $\mathsf{H}^{(n-2)}(u,h) < \mathsf{H}^{(n-1)}(v,h)$ and $\mathsf{H}^{(n-1)}(u,h) < \mathsf{H}^{(n-1)}(v,h)$, u will not trigger the decrease of $\mathsf{H}^{(n)}(v,h)$.*

According to Lemma 4.1, (1) if all of the h-neighbors of v do not change their $(n-1)$-order H-indexes, $\mathsf{H}^{(n)}(v,h)$ remains unchanged; (2) If a vertex u's $(n-1)$-order H-index decreases, it may affect the n-order H-index of u's h-neighbor v. But, in the following two cases, u does not trigger the decrease of $\mathsf{H}^{(n)}(v,h)$ compared with $\mathsf{H}^{(n-1)}(v,h)$: (a) both the $(n-2)$-order H-index and $(n-1)$-order H-index of u are larger than the $(n-1)$-order H-index of v; (b) both the $(n-2)$-order H-index and $(n-1)$-order H-index of u are smaller than the $(n-1)$-order H-index of v.

Based on Lemma 4.1, we propose the second optimization for reducing redundant computations. Specifically, in Algorithm 1, if the n-order H-index of u does not satisfy neither one of the two cases in Lemma 4.1, it needs to make marks to compute $\mathsf{H}^{(n+1)}(v,h)$ for $v \in N_G(u,h)$ in the next iteration. Thus, for those vertices that are not marked, we can save the computation of $\mathsf{H}^{(n+1)}(v,h)$ by directly copying from $\mathsf{H}^{(n)}(v,h)$ to $\mathsf{H}^{(n+1)}(v,h)$.

Example 3. *We integrate Optimization 2 into Algorithm 1 and run it on the graph in Figure 5. The local algorithm + Optimization 2 in Table 1 shows the computation of n-order H-indexes, where 1 denotes that the n-order H-index is computed and 0 denotes not. Optimization 2 can reduce unnecessary computations by* 37.5%.

Optimization 3: Lazy n-order H-index.

As analyzed in Section 4.1, Algorithm 2 takes $O(h(|\tilde{V}| + |\tilde{E}|))$ time for computing $\mathsf{H}^{(n)}(v,h)$. If there is a large number of such n-order H-index computations, it should be prohibitively expensive. Thus, the third optimization aims to reduce the number of n-order H-index computations. To this end, we introduce a new concept called *lazy n-order H-index*.

Definition 4.1. *Given a graph G and a positive integer h, the lazy n-order H-index of v, denoted by $\mathsf{H}_G^{*(n)}(v,h)$, is*

$$\mathsf{H}_G^{*(n)}(v,h) = \begin{cases} \deg_G(v,h) & n = 0 \\ \mathcal{H}(\mathsf{H}_G^{*(n-1)}(u_1,h), \ldots, \mathsf{H}_G^{*(n-1)}(u_i,h)) & n > 0 \end{cases} \quad (10)$$

where $i = \deg_G(v,h)$ and $\forall j \in [1,i]$, $u_j \in \mathsf{N}_G(v,h)$.

The lazy n-order H-index directly uses $(n-1)$-order H-index, instead of $(n-1)$-order attainability index in Def. 3.2. In the following, we analyze the properties of lazy n-order H-index.

Lemma 4.2. *Given a graph G and a positive integer h, $\forall v \in V_G$ and $\forall n \in \mathbb{N}$, it holds that $\mathsf{H}^{*(n)}(v,h) \geq \mathsf{H}^{*(n+1)}(v,h)$.*

In other words, $\mathsf{H}^{*(n)}(v,h)$ is monotonously non-increasing w.r.t. the increasing n, i.e., $\mathsf{H}^{*(0)}(v,h) \geq \mathsf{H}^{*(1)}(v,h) \geq \ldots$ holds. As with $\mathsf{H}^{(n)}(v,h)$, when the number n is large enough, $\mathsf{H}^{*(n)}(v,h)$ also can converge to *a nonnegative value*.

Theorem 4.1.

$$\lim_{n\to\infty} \mathsf{H}^{*(n)}(v,h) \geq \mathsf{C}(v,h). \quad (11)$$

Proof. First, we prove that $\forall v \in V_G$ and $\forall n \in \mathbb{N}$, $\mathsf{H}^{*(n)}(v,h) \geq \mathsf{H}^{(n)}(v,h)$. We can prove it by mathematical induction.

(i) For $n = 0$, $\forall v \in V_G$, $\mathsf{H}^{*(0)}(v,h) = \mathsf{H}^{(0)}(v,h) = \deg_G(v,h)$.

(ii) Assume that the result is valid for $n = m$, i.e., $\forall v \in V_G$, $\mathsf{H}^{*(m)}(v,h) \geq \mathsf{H}^{(m)}(v,h)$. Based on Equation 3, $\mathcal{A}_v^{(m)}(u,h) \leq \mathsf{H}^{(m)}(u,h) \leq \mathsf{H}^{*(m)}(u,h)$. For $\mathsf{H}^{(m+1)}(v,h)$, we have

$$\begin{aligned}\mathsf{H}^{(m+1)}(v,h) &= \mathcal{H}(\mathcal{A}_v^{(m)}(u_1,h), ..., \mathcal{A}_v^{(m)}(u_i,h)) \\ &\leq \mathcal{H}(\mathsf{H}^{(m)}(u_1,h), ..., \mathsf{H}^{(m)}(u_i,h)) \\ &\leq \mathcal{H}(\mathsf{H}^{*(m)}(u_1,h), ..., \mathsf{H}^{*(m)}(u_i,h)) \\ &= \mathsf{H}^{*(m+1)}(v,h)\end{aligned}$$

Hence, it also holds for $n = m + 1$. As a result, $\mathsf{H}^{*(n)}(v,h) \geq \mathsf{H}^{(n)}(v,h)$ holds.

When $n \to \infty$, $\mathsf{H}^{*(\infty)}(v,h) \geq \mathsf{H}^{(\infty)}(v,h) = \mathsf{C}(v,h)$. Thus, the theorem is proved. □

Based on Theorem 4.1, the lazy n-order H-index of vertex v is an upper bound of its coreness $\mathsf{C}(v,h)$, even when n becomes infinity. If we use $\mathsf{H}^{*(\infty)}(v,h)$ to initialize $\mathsf{H}^{(0)}(v,h)$ in Algorithm 1, it can help reduce the iterations since $\mathsf{H}^{*(\infty)}(v,h) \leq \deg_G(v,h)$. Fortunately, $\mathsf{H}^{(n)}(v,h)$ can still converge to its coreness, thanks to Theorem 3.5. It is worth mentioning that the total number of iterations (the number of iterations for lazy n-order H-index computation plus the number of iterations for n-order H-index computation) may increase, but the overall performance of Algorithm 1 using Optimization 3 improves. This is because the computation of lazy n-order H-index only takes $O(|\tilde{V}|)$ time, which is more efficient than that of n-order H-index($O(h(|\tilde{V}| + |\tilde{E}|))$). Thus, this optimization of lazy n-order H-index can improve the performance of our local algorithm significantly.

Example 4. *Table 1 shows the improvement of Algorithm 1 implemented by Optimization 3 of lazy n-order H-index. The algorithm first computes the lazy n-order H-index for every vertex. After three iterations, the lazy n-order H-indexes of all vertices have converged. Then, it continues to perform the n-order H-index computation. It takes only one iteration to converge and obtain the results, which use two less iterations than Algorithm 1.*

5 (k,h)-CORE MAINTENANCE OVER DYNAMIC GRAPHS

In the previous sections, we have studied and proposed algorithms for (k,h)-*core decomposition in static graphs*. Differently, in this section, we further investigate the problem of (k,h)-*core maintenance in dynamic graphs*, where the network structure changes with edge insertions and deletions.

5.1 (k,h)-Core Maintenance

In many real-word applications, a graph undergoes constant updates with edge insertions/deletions. The problem of (k,h)-core maintenance over a dynamic graph G focuses on the coreness update when an edge (u,v) is inserted into or deleted from G. Note that our proposed algorithms can be easily extended to handle a batch of edge insertions and deletions, as well as node insertions/deletions, because the node insertion/deletion can be simulated by a sequence of edge insertion and deletion preceded/followed by an insertion/deletion of a node.

In the remainder of this section, we consistently use $\mathsf{C}_G(w,h)$ and $\mathsf{C}_{G'}(w,h)$ to represent the coreness of a vertex w before and after an edge insertion/deletion of (u,v), respectively, where G' is a new graph updated from G. We have the following properties.

Lemma 5.1. *Given a vertex $w \in V(G) \cup V(G')$,*

1. *if $N_G(w,h) \neq N_{G'}(w,h)$, the coressness of w may change, i.e., $\mathsf{C}_G(w,h) \neq \mathsf{C}_{G'}(w,h)$;*
2. *if $\mathsf{C}_G(w,h) \neq \mathsf{C}_{G'}(w,h)$, the coressness of w's h-neighbor $w' \in N_G(w,h) \cup N_{G'}(w,h)$ may change;*
3. *if $\mathsf{C}_G(w,h) \neq \mathsf{C}_{G'}(w,h)$, then $|\mathsf{C}_G(w,h) - \mathsf{C}_{G'}(w,h)| \geq 1$ holds.*

For $h = 1$, the problem of (k,h)-core maintenance is exactly equivalent to the problem of core maintenance [39]. Existing studies [29, 39, 40] show that $|\mathsf{C}_G(w,h) - \mathsf{C}_{G'}(w,h)| = 1$ for core maintenance. However, in our (k,h)-core maintenance for $h > 1$, the change of $|\mathsf{C}_G(w,h) - \mathsf{C}_{G'}(w,h)|$ may be much larger than 1. This leads to a more challenging task for (k,h)-core maintenance. An intuitive method to address the (k,h)-core maintenance is to recompute the coreness for every vertex in the updated graph G by Algorithm 1 or peeling methods [9]. Obviously, these computing-from-scratch approaches are inefficient when the graph G is large and the graph updates happen frequently.

In this section, we propose efficient algorithms by extending our proposed local algorithm in Algorithm 1. Recall that Algorithm 1 iteratively computes the n-order H-index from the initial h-degree for every vertex until it converges. Actually, in most cases, there exist a number of vertices that keep their corenesses unchanged, i.e., $\mathsf{C}_G(w,h) = \mathsf{C}_{G'}(w,h)$. Hence, it is unnecessary to recompute the n-order H-index for all vertices. In addition, the obtained coreness $\mathsf{C}_G(w,h)$ in the original graph G can be used for the (k,h)-core computation in the updated graph G'. Motivated by this, when we extend the local algorithm to handle the (k,h)-core maintenance, we need to consider two key questions, i.e., (1) how to identify the unaffected vertices in G' precisely; and (2) how to best utilize the existing coreness in G.

5.2 Edge Deletion Algorithm

We present an edge deletion algorithm for (k,h)-core maintenance based on two important rules as follows.

Identification of Unaffected Vertices. We first identify those vertices having unchanged corenesses when an edge (u,v) is deleted from graph G.

Theorem 5.1. *For a removed edge $e = (u,v)$ and a vertex w in G, if $\mathsf{C}_G(w,h) > \min\{\mathsf{C}_G(u,h), \mathsf{C}_G(v,h)\}$, it holds that $\mathsf{C}_G(w,h) = \mathsf{C}_{G'}(w,h)$.*

Based on Theorem 5.1, if a vertex's coreness in the original graph G is larger than $\min\{\mathsf{C}_G(u,h), \mathsf{C}_G(v,h)\}$, its coreness does not change in the updated graph G'.

Utilization of Original Corenesses. Next, we explore the utilization of the existing corenesses.

Theorem 5.2. *For any subgraph $G' \subseteq G$ and a vertex $v \in V_{G'}$, $\mathsf{C}_G(v,h) \geq \mathsf{C}_{G'}(v,h)$ holds.*

Algorithm 3 Edge Deletion Algorithm

Input: a graph G; a positive integer h; a deleted edge $e = (u, v) \in E_G$; original corenesses $\{C_G(w, h) : w \in V(G)\}$
Output: new corenesses $\{C_{G'}(w, h) : w \in V(G')\}$;
1: Let be a new graph G' with $V_{G'} = V_G$ and $E_{G'} = E_G \setminus \{(u, v)\}$;
2: **for** each vertex $w \in V_{G'}$ **do**
3: **if** $C_G(w, h) > \min(C_G(u, h), C_G(v, h))$ **then**
4: $Update[w] \leftarrow$ **false**;
5: $H^{(0)}[w] \leftarrow C_G(w, h)$;
6: **else**
7: $Update[w] \leftarrow$ **true**;
8: Compute $\deg_{G'}(w, h)$;
9: $H^{(0)}[w] \leftarrow \min(C_G(w, h), \deg_{G'}(w, h))$;
10: Iteratively compute $H^{(n)}[w]$ for $w \in V_{G'}$ with $Update[w]$ = **true** until it converges; // Invoking the lines 4-11 of Algorithm 1
11: $C_{G'}(v, h) \leftarrow H^{(n)}[v]$ for each vertex $v \in V_G$;
12: **return** the new corenesses $\{C_{G'}(v, h) : v \in V_{G'}\}$;

Theorem 5.2 indicates that v's coreness in graph G is no smaller than its coreness in any updated graph $G' \subseteq G$. Hence, $C_G(v, h)$ is an upper bound of $C_{G'}(v, h)$. According to the relative independence of 0-order H-index presented in Theorem 3.5, we use $C_G(v, h)$ to initialize the 0-order H-index of v in G'. If $C_G(v, h)$ is smaller than its h-degree in the updated graph, $|deg_{G'}(v, h)|$, this initialization strategy can help the algorithm converge in a faster way. Note that if $C_G(v, h)$ is larger than $|deg_{G'}(v, h)|$, we still use $|deg_{G'}(v, h)|$ to initialize the 0-order H-index.

Based on the above two useful rules, we propose an edge deletion algorithm for (k, h)-core maintenance in Algorithm 3. Specifically, the algorithm first identifies the vertices that do not change their corenesses in G' (lines 3-5) and also initializes the 0-order H-index for the affected vertices (lines 7-9). Then, it iteratively computes the n-order H-index for all affected vertices until they converge, which uses the same steps of Algorithm 1 and omits the duplicated details (line 10). Finally, the algorithm returns the new corenesses $\{C_{G'}(v, h) : v \in V_{G'}\}$.

5.3 Edge Insertion Algorithm

In this section, we present an edge insertion algorithm for (k, h)-core maintenance. We first analyze the identification of unaffected vertices.

Identification of Unaffected Vertices. Based on Lemma 5.1, an edge insertion may increase the corenesses of some vertices, which in turn increase the corenesses of their h-neighbors. But, we find that some of the h-neighbors do not change their corenesses. Specifically, assume that for two vertices w and w', the coreness of w increases after an edge insertion, and $w' \in N_{G'}(w, h)$ holds. If $C_G(w, h)$ is larger than $C_G(w', h)$, the coreness of w' will not increase in the updated G'. Accordingly, we have the following updating rule.

Theorem 5.3. *Given a graph G, G has an update with an edge insertion of $(u, v) \notin E_G$. Defining a vertex set $S = N_G(u, h-1) \cup N_G(v, h-1) \cup \{u, v\}$. If $C_G(w, h) < \min_{w' \in S} C_G(w', h)$, it holds that $C_G(w, h) = C_{G'}(w, h)$.*

Theorem 5.3 indicates that the vertices, whose corenesses are smaller than the minimum coreness of the h-degree changed vertices in the new graph G', keep their corenesses unchanged.

Algorithm 4 Edge Insertion Algorithm

Input: a graph G; a positive integer h; an inserted edge $e = (u, v) \notin E_G$; original corenesses $\{C_G(w, h) : w \in V(G)\}$
Output: new corenesses $\{C_{G'}(w, h) : w \in V(G')\}$;
1: Let be a new graph G' with $V_{G'} = V_G$ and $E_{G'} = E_G \cup \{(u, v)\}$;
2: Let be a vertex set: $S \leftarrow N_G(u, h-1) \cup N_G(v, h-1) \cup \{u, v\}$;
3: A lower bound: $k_{min} \leftarrow \min_{w \in S} C(w, h)$;
4: A vertex set: $S'' \leftarrow \{w'' \in V_{G'} : C(w'', h) \geq k_{min}\}$
5: Let G'' be a subgraph of G' induced by S'';
6: **for** each vertex $w \in V_{G'}$ **do**
7: **if** $C(w, h) < k_{min}$ **then**
8: $Update[w] \leftarrow$ **false**;
9: $H^{(0)}[w] \leftarrow C(w, h)$;
10: **else**
11: $Update[w] \leftarrow$ **true**;
12: Compute $\deg_{G''}(w, h)$;
13: $H^{(0)}[w] \leftarrow \deg_{G''}(w, h)$;
14: Iteratively compute $H^{(n)}[w]$ for $w \in V_{G'}$ with $Update[w]$ = **true** until it converges; // Invoking the lines 4-11 of Algorithm 1
15: $C_{G'}(v, h) \leftarrow H^{(n)}[v]$ for each vertex $v \in V_G$;
16: **return** the new corenesses $\{C_{G'}(v, h) : v \in V_{G'}\}$;

Utilization of Original Corenesses. Next, we explore the utilization of the existing corenesses in G.

Theorem 5.4. *Give a new graph G' after an edge insertion of (u, v). Let G'' be a subgraph of G' induced by the vertex set V_S, where $V_S = \{w \in V_G : C_G(w, h) \geq \min_{w' \in S} C_G(w', h)\}$ and $S = N_G(u, h-1) \cup N_G(v, h-1) \cup \{u, v\}$. It holds that $\forall w \in V_{G''}$, $C_{G'}(w, h) \leq \deg_{G''}(w, h)$.*

Proof. Since the vertices in $V_{G'} \setminus V_{G''}$ have smaller corenesses than that of all vertices in $V_{G''}$, these vertices does not affect the coreness of vertices in $V_{G''}$. In other words, $\forall w \in V_{G''}$, $C_{G'}(w, h) = C_{G''}(w, h)$. In addition, $\forall w \in V_{G''}$, $C_{G''}(w, h) \leq \deg_{G''}(w, h)$. Hence, $\forall w \in V_{G''}$, $C_{G'}(w, h) \leq \deg_{G''}(w, h)$. □

Theorem 5.5. *For a vertex $w \in V_G$, if $\exists n \in \mathbb{N}$ such that $H^{(n)}_{G'}(w, h) = C_G(w, h)$, it holds that $C_{G'}(w, h) = C_G(w, h)$.*

Both Theorems 5.4 and 5.5 can be used to accelerate the n-order H-index computation for affected vertices. Specifically, Theorem 5.4 can use $\deg_{G''}(w, h)$ as 0-order H-index for initialization, as $\deg_{G''}(w, h) \leq \deg_{G'}(w, h)$. Theorem 5.4 can reduce the n-order H-index computation redundancy for the vertices when their corenesses have converged.

Algorithm. Based on above three theorems, we propose an edge insertion algorithm for (k, h)-core maintenance in Algorithm 4. Specifically, the algorithm first computes a lower bound k_{min} (lines 2-3), i.e., the minimum affected coreness, and forms the subgraph G' induced by the vertices whose coreness in the origin graph G is no smaller than k_{min} (lines 4-5). Then, it identifies all unaffected vertices based on Theorem 5.3 (lines 7-9), and sets the 0-order H-index for the affected vertices by Theorem 5.4 (lines 11-13). Next, it iteratively computes the n-order H-index for all affected vertices until it converges (line 14). It is worth mentioning that in these iterations, the algorithm employs Theorem 5.5 to stop the n-order H-index computation of v when v's n-order H-index in G' reaches its coreness $C_G(v, h)$ in G. Finally, the algorithm returns the result of new corenesses $\{C_{G'}(v, h) : v \in V_{G'}\}$.

Table 2: Dataset Statistics (d_{avg}, d_{max}, $diam$ represent the average degree, the maximum degree, and the network diameter, respectively; K = 10^3, M = 10^6, and B = 10^9)

Dataset	Abbr.	$\|V_G\|$	$\|E_G\|$	d_{avg}	d_{max}	$diam$
celegans	CE	453	2K	8	237	8
dmela	DM	7K	26K	6	190	12
Facebook	FB	4K	88.2K	44	1K	8
ca-HepPh	CH	11.2K	118K	20	491	13
WormNet	WN	15.2K	246K	33	375	13
Douban	DB	155K	327K	4	287	9
Amazon	AM	335K	926K	3	549	44
DBLP	DP	317K	1M	6	343	21
roadNet-PA	PA	1.1M	1.5M	3	9	786
Flickr	FK	106K	2.3M	44	5.4K	9
Hyves	HY	1.4M	2.8M	4	31K	10
Youtube	YT	1.1M	2.99M	5	29K	20
road-asia	RA	12M	13M	2	9	48K
Patent	PT	3.7M	16.5M	8	793	22
road-usa	RU	24M	29M	2	9	256
Orkut	OR	3M	117M	76	33K	9
uk-2005	UK	39M	783M	40	1.7M	23
it-2004	IT	41M	1.1B	50	1.3M	26

6 EMPIRICAL EVALUATION

We conduct experiments to evaluate the efficiency of our proposed algorithms. All experiments are implemented in C++ and conducted on a Linux Server with 2.10 GHz CPU and 256 GB memory running Ubuntu. The algorithms are parallelized using OpenMP. All algorithms use 16 threads for parallel computation by default.

Datasets. We use 18 different types of real-world networks in experiments. Table 2 summarizes the statistics of these networks. Specifically, celegans[3], dmela[3], and WormNet[3] are biological networks; ca-HepPh[4] and DBLP[4] are collaboration networks; Amazon[4] is a co-purchasing network; Patent[4] is a citation network; roadNet-PA[4], roadNet-asia[3], and roadNet-usa[3] are road networks; Douban[3], Facebook[4], Youtube[4], Flickr[4], Orkut[4] , and Hyves[5] are social networks; uk-2005[3] and it-2004[3] are web graphs.

6.1 Convergence Evaluation

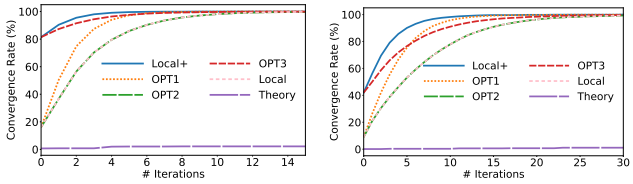
In this experiment, we conduct the convergence evaluation of our algorithms on the real-world networks. In Section 3, we show that a vertex's n-order H-index can finally converge to its coreness and offer a theoretical upper bound for the number of iterations by Theorem 3.3. We compare five of our algorithms including the local algorithm in Algorithm 1 (*Local*), Algorithm 1 integrating our three optimizations respectively, i.e., *Local+OPT1*, *Local+OPT2*, and *Local+OPT3*, and Algorithm 1 integrating all three optimizations (*Local+*), against the theoretical upper bound (*Theory*).

Exp-1: Evaluation of the number of iterations. Table 3 reports the number of iterations required by different algorithms to converge for $h = 2$ and $h = 3$. Note that Optimization 3 uses the lazy n-order H-index $\mathrm{H}^{*(n)}(\cdot)$. In Table 3, we also report the number of iterations for computing $\mathrm{H}^{*(n)}(\cdot)$. We can observe that the number

[3]http://networkrepository.com/
[4]http://snap.stanford.edu/
[5]http://konect.cc/networks/

Table 3: # Iterations of Different Methods

h	Methods		Datasets				
			CE	FB	DM	AM	PA
$h = 2$	Theory		128	221	1,270	30,511	11,436
	Local		12	14	42	35	188
	Local+OPT1		7	8	24	18	99
	Local+OPT2		12	14	42	35	188
	Local+OPT3	$\mathrm{H}^{*(n)}(\cdot)$	12	14	44	29	136
		$\mathrm{H}^{(n)}(\cdot)$	6	1	40	31	172
	Local+	$\mathrm{H}^{*(n)}(\cdot)$	8	8	23	14	85
		$\mathrm{H}^{(n)}(\cdot)$	**6**	**1**	**22**	**15**	**97**
$h = 3$	Theory		45	27	2,667	52,323	68,392
	Local		7	7	39	72	416
	Local+OPT1		5	6	21	38	208
	Local+OPT2		7	7	39	72	416
	Local+OPT3	$\mathrm{H}^{*(n)}(\cdot)$	7	7	55	54	253
		$\mathrm{H}^{(n)}(\cdot)$	1	2	36	69	323
	Local+	$\mathrm{H}^{*(n)}(\cdot)$	5	6	29	36	143
		$\mathrm{H}^{(n)}(\cdot)$	**1**	**2**	**19**	**30**	**153**

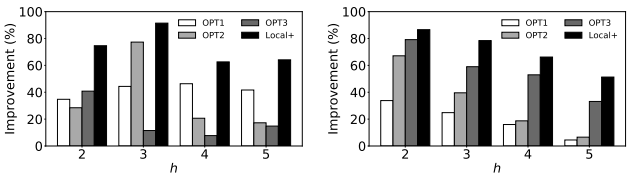


(a) $h = 2$ (b) $h = 3$

Figure 6: Convergence Rate on Amazon

of iterations required by our algorithms is much less than that of Theory. For example, for Local, when $h = 2$, it converges within 4.2% of the total iterations needed by Theory on average. Recall that Optimizations 1 and 3 use asynchronous processing and lazy n-order H-index to optimize the local algorithm. Both of them can reduce the number of iterations. On the other hand, Optimization 2 mainly tries to reduce the redundant n-order H-index computation in each iteration, which does not affect the number of iterations. Thus, both Local+OPT1 and Local+OPT3 have less iterations than Local while Local+OPT2 has the same number of iterations as Local. Overall, our optimized algorithm of Local+ consumes the least iterations since it integrates all three optimizations.

Exp-2: Evaluation of convergence rate. We evaluate the convergence rate of all algorithms, from which we can see how fast each algorithm converges. The convergence rate of an iteration is computed as the percentage of the accumulated converged vertices when this iteration finishes. When the convergence rate is equal to 100%, all vertices converge. Figures 6(a) and 6(b) show the results of convergence rate on Amazon for $h = 2$ and $h = 3$, respectively. We can observe that our algorithms converge much faster than Theory. For example, at the 10-th iteration, almost 97% of the vertices have converged for Local while only 2% of the vertices have converged in Theory. Moreover, for our algorithms, the majority of the vertices can converge in a few iterations. As shown in Table 3, the number of iterations of Local on Amazon is 35 for $h = 2$. Figure 6(a) shows that Local's convergence rate has reached 97% in 10 iterations (less than $\frac{1}{3}$ of the total iterations). Moreover, Local+ achieves the best converge performance among all methods.



(a) CH (b) FB

Figure 7: Efficiency of optimizations

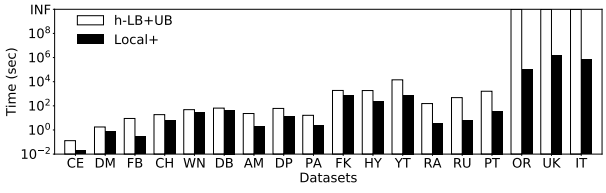


Figure 8: Local+ VS. h-LB+UB

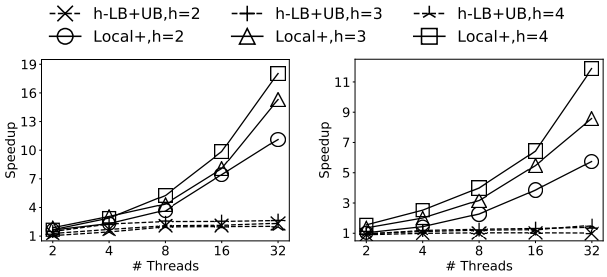
6.2 Efficiency Evaluation

In this experiment, we evaluate the efficiency of (k, h)-core decomposition algorithms. Note that we denote the running time as 'INF' if an algorithm cannot complete within two weeks.

Exp-3: Evaluation of optimization efficiency. We evaluate the efficiency of the three optimizations proposed in Section 4.2 on CH and FB. We vary h from 2 to 5. There are four competitor methods including Optimization 1 (OPT1), Optimization 2 (OPT2), Optimization 3 (OPT3), and the combination of all of them (Local+). Figure 7 reports the efficiency improvement results of all different optimization methods w.r.t. the local algorithm in Algorithm 1. We can observe that all the three optimizations are effective and efficient. For example, when $h = 2$, OPT1, OPT2, OPT3, and Local+ improve the performance of the local algorithm by up to 34%, 67%, 79%, and 87% on FB dataset, respectively. On average, OPT1, OPT2, OPT3, and Local+ improve the performance of the local algorithm by up to 42%, 36%, 19%, and 73% on CH dataset, respectively. In addition, we can observe that for FB, the improvements of optimizations decrease with the increase of h. This is because the larger h, the *denser* the graph, which deteriorates the effect of optimizations.

Exp-4: Local+ VS. h-LB+UB. We next compare the efficiency performance of our optimized method Local+ with h-LB+UB [9], which is the state-of-the-art algorithm for (k, h)-core decomposition. Note that we implement h-LB+UB following the pseudocodes outlined in [9], but make minor revisions to obtain the exact results for all cases. We make two modifications. Specifically, (i) at line 6 of Algorithm 3 [9], we move v to $B[\max(\deg_{G[V\setminus v]}(u, h), k)]$; and (ii) at line 13 of Algorithm 6 [9], we compute the exact h-degree for the vertices. The running time results are reported in Figure 8. Local+ consistently outperforms h-LB+UB, by 1-3 orders of magnitude, on all datasets. Moreover, h-LB+UB cannot finish within two weeks on OR, UK, and IT, whereas Local+ still works well for these datasets, which validates the superiority of our proposed algorithm. Note that the performance of Local+ can be further improved by using more threads, thanks to its high parallelism, as will be shown in the next experiment.

Exp-5: Parallelism evaluation w.r.t. # threads. We test the parallelism performance of both Local+ and h-LB+UB varying by the number of threads. Figure 9 reports the results by varying the number of threads from 1 to 32. Obviously, using more threads allows more vertices to compute the n-order H-indexes simultaneously in a single iteration and thus improves the overall performance of Local+. For example, in Figure 9(a), if $h = 4$, Local+ is 18x faster when the number of threads increases from 1 to 32. In addition, Local+ has higher parallelism performance than h-LB+UB. Moreover, we can observe that in some cases the speedup of Local+ is not very high. For example, in Figure 9(b), if $h = 2$, the speedup of Local+ is 5 on AM when the number of threads increases to 32. The limited speedup is due to two reasons: 1) As shown in Exp-1, most vertices converge within the first few iterations. In the remaining iterations, only a few vertices compute the n-order H-index in parallel, which limits the effect of parallelism. 2) The vertices with a high h-degree take a long time to compute the H-index, which locks a thread for a prolonged time.



(a) FB (b) AM

Figure 9: Parallelism evaluation w.r.t. # threads

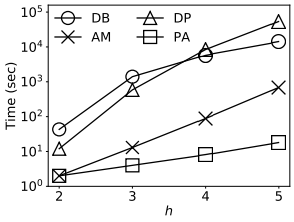


Figure 10: Effect of h

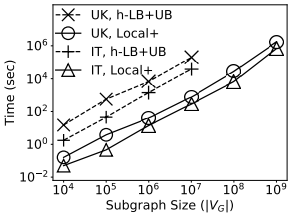


Figure 11: Effect of $|V_G|$

Exp-6: Effect of h. We evaluate the effect of h on Local+. Figure 10 shows the results of Local+ by varying h from 2 to 5 on four datasets. With the increase of h, the performance of Local+ becomes worse. It is because when h increases, the size of a vertex's h-neighbors enlarges, which leads to two efficiency issues: (1) the computation of n-order H-index takes more time since the subgraph induced by a vertex's h-neighbors becomes larger; and (2) the total number of iterations may increase since the value of 0-order H-index increases.

Exp-7: Effect of $|V_G|$. We evaluate the effect of $|V_G|$ using a set of subgraphs extracted from the large datasets UK and IT with different fractions of vertices. The results are reported in Figure 11. As shown, when the graph size increases, Local+ takes more time for (k, h)-core decomposition. Nevertheless, (i) Local+ is consistently much faster than h-LB+UB on all sizes of graphs; (ii) h-LB+UB cannot complete within two weeks when the graph size is larger than 10^8 while Local+ scales better to large graphs.

Table 4: Update Time for Edge Deletions (sec)

Update Mode	h	Methods	Datasets				
			FB	DM	CH	AM	PA
Single	$h=2$	Local+	0.26	0.83	5.57	1.89	2.23
		EDA	0.11	0.01	0.09	0.01	0.01
	$h=3$	Local+	0.91	9.35	7.44	14.16	4.37
		EDA	0.64	0.66	1.08	0.01	0.01
Batch	$h=2$	Local+	0.29	0.88	5.78	2.08	2.22
		EDA	0.26	0.18	0.92	0.02	0.01
	$h=3$	Local+	0.92	10.02	7.44	13.14	4.32
		EDA	0.89	4.86	4.92	0.3	0.01

Table 5: Update Time for Edge Insertions (sec)

Update Mode	h	Methods	Datasets				
			FB	DM	CH	AM	PA
Single	$h=2$	Local+	0.23	0.77	5.82	2.01	2.37
		EIA	0.01	0.59	4.01	0.54	0.79
	$h=3$	Local+	0.95	9.15	7.44	14.09	4.32
		EIA	0.37	8.2	3.26	9.32	2.03
Batch	$h=2$	Local+	0.26	0.85	5.47	2.01	2.23
		EIA	0.08	0.69	4.75	0.67	0.86
	$h=3$	Local+	1.2	9.35	7.45	14.09	4.45
		EIA	1.18	9.24	5.42	9.75	2.21

6.3 Update Evaluation

In this experiment, we evaluate the efficiency of our proposed updating algorithms for (k,h)-core maintenance over dynamic graphs, including the edge deletion algorithm in Algorithm 3 (EDA) and the edge insertion algorithm in Algorithm 4 (EIA). We compare the algorithms with Local+, which recomputes (k,h)-core using the optimized local algorithm. Two update modes are considered, i.e., single edge update and a batch of edge updates. For the batch updates, we insert or delete 2–100 edges at a time. In each experiment, we generate 500 updates and report the average results using 16 threads. Note that the edges for updating are randomly generated.

Exp-9: Efficiency of edge deletion update. We evaluate the efficiency of the edge deletion algorithms on five datasets. The results are reported in Table 4. We can observe that EDA is much faster than Local+, achieving 1 to 2 orders of magnitude of improvement in most cases. However, the advantage of EDA for the batch update is not as great as the single edge update. This is because the batch processing usually makes more vertices to change their h-corenesses. Hence, EDA takes more time. But the performance of Local+ does not deteriorate too much since the cardinality of the graph is only slightly changed.

Exp-10: Efficiency of edge insertion update. Finally, we study the efficiency of edge insertion updating. Table 5 shows the results for EIA and Local+ in terms of single and batch processing under $h=2$ and $h=3$. As expected, EIA has a better performance than Local+ in all cases. For example, when $h=2$, EIA is 50% and 38.2% faster than Local+ for single and batch processing, respectively.

7 RELATED WORK

Core Decomposition. Due to widespread applications, such as graph visualization [2, 5], community search [14, 25, 42], network topology analysis [3, 10], and system structure analysis [50], the problem of core decomposition has been extensively studied [33]. An in-memory algorithm [6], a disk-based algorithm [11], and a semi-external algorithm [48] are proposed for core decomposition. Khaouid et al. [28] implemented the algorithms of [6, 11] based on the GraphChi and Webgraph models. In addition, several local algorithms [32, 37, 41] have been proposed for core decomposition by utilizing the vertices' neighborhood information. However, these local algorithms do not consider the (k,h)-attainability of (k,h)-core and thus cannot be used for (k,h)-core decomposition. Based on different computing architectures, many works proposed parallel algorithms [15, 17] and distributed algorithms [34, 36]. Besides, core decomposition has also been studied over directed graphs [22], weighted graphs [16], bipartite graphs [30], temporal graphs [21, 49], uncertain graphs [8, 38], multilayer graphs [19, 20], and heterogeneous information networks [18].

Core Maintenance. The problem of core maintenance is to update vertices' corenesses when the graph evolves over time. Sariyuce et al. [39, 40] and Li et al. [29] developed several incremental algorithms. Zhang et al. [51] proposed a new order-based algorithm through explicitly maintaining the k-order for a graph. Note that the k-core maintenance algorithms proposed by [29, 39, 40, 51] simulate the peeling-based k-core decomposition algorithms. However, these algorithms usually resort to auxiliary structures, e.g., [51] needs to keep track of the vertex deletion order, to maintain the vertex's degree, which yields additional space complexity. It is also non-trivial to extend these algorithms to (k,h)-core maintenance due to the (k,h)-attainability of (k,h)-core. More recently, semi-external algorithms [48], distributed algorithms [1], and parallel algorithms [24, 27, 46] are proposed for core maintenance. In addition, Bai et al. [4] studied the core maintenance problem for temporal graphs.

In contrast to the k-core that only considers a vertex's 1-neighbors, the (k,h)-core takes a vertex's h-neighbors into consideration, which is more complex [9]. New (k,h)-core decomposition and maintenance algorithms are proposed in this paper to address the deficiency of the state-of-the-art algorithms in [9].

8 CONCLUSIONS

This paper revisits the problem of (k,h)-core decomposition. First, we introduce two novel concepts of pairwise h-attainability index and n-order H-index. We show that the n-order H-index possesses several nice properties such as convergence and asynchrony. Based on that, we propose a parallelizable local algorithm for (k,h)-core decomposition, which iteratively computes each vertex's n-order H-index until convergence. Several optimizations are developed to further enhance the efficiency of the local algorithm. Moreover, we extend the local algorithm to handle (k,h)-core maintenance over dynamic graphs. Finally, experimental results based on real-world datasets demonstrate the efficiency of our proposed algorithms.

ACKNOWLEDGMENTS

This work is supported by Research Grants Council of Hong Kong under GRF/ECS Projects 22200320, 12200819, 12200917, CRF Project C6030-18GF, and Guangdong Basic and Applied Basic Research Foundation (Project No. 2019B1515130001). Jianliang Xu is the corresponding author.

REFERENCES

[1] Hidayet Aksu, Mustafa Canim, Yuan-Chi Chang, Ibrahim Korpeoglu, and Özgür Ulusoy. 2014. Distributed k-Core View Materializationand Maintenance for Large Dynamic Graphs. *IEEE Trans. Knowl. Data Eng.* 26, 10 (2014), 2439–2452.

[2] J. Ignacio Alvarez-Hamelin, Luca Dall'Asta, Alain Barrat, and Alessandro Vespignani. 2005. Large scale networks fingerprinting and visualization using the k-core decomposition. In *NIPS*. 41–50.

[3] José Ignacio Alvarez-Hamelin, Luca Dall'Asta, Alain Barrat, and Alessandro Vespignani. 2008. K-core decomposition of Internet graphs: hierarchies, self-similarity and measurement biases. *Networks Heterog. Media* 3, 2 (2008), 371–393.

[4] Wen Bai, Yadi Chen, and Di Wu. 2020. Efficient temporal core maintenance of massive graphs. *Inf. Sci.* 513 (2020), 324–340.

[5] Vladimir Batagelj, Andrej Mrvar, and Matjaz Zaversnik. 1999. Partitioning Approach to Visualization of Large Graphs. In *GD*. 90–97.

[6] Vladimir Batagelj and Matjaz Zaversnik. 2003. An O(m) Algorithm for Cores Decomposition of Networks. *CoRR* cs.DS/0310049 (2003). http://arxiv.org/abs/cs/0310049

[7] Devora Berlowitz, Sara Cohen, and Benny Kimelfeld. 2015. Efficient Enumeration of Maximal k-Plexes. In *SIGMOD*. 431–444.

[8] Francesco Bonchi, Francesco Gullo, Andreas Kaltenbrunner, and Yana Volkovich. 2014. Core decomposition of uncertain graphs. In *SIGKDD*. 1316–1325.

[9] Francesco Bonchi, Arijit Khan, and Lorenzo Severini. 2019. Distance-generalized Core Decomposition. In *SIGMOD*. 1006–1023.

[10] Shai Carmi, Shlomo Havlin, Scott Kirkpatrick, Yuval Shavitt, and Eran Shir. 2007. A model of Internet topology using k-shell decomposition. *Proceedings of the National Academy of Sciences* 104, 27 (2007), 11150–11154.

[11] James Cheng, Yiping Ke, Shumo Chu, and M. Tamer Özsu. 2011. Efficient core decomposition in massive networks. In *ICDE*. 51–62.

[12] James Cheng, Yiping Ke, Ada Wai-Chee Fu, Jeffrey Xu Yu, and Linhong Zhu. 2010. Finding maximal cliques in massive networks by H*-graph. In *SIGMOD*. 447–458.

[13] James Cheng, Linhong Zhu, Yiping Ke, and Shumo Chu. 2012. Fast algorithms for maximal clique enumeration with limited memory. In *SIGKDD*. 1240–1248.

[14] Wanyun Cui, Yanghua Xiao, Haixun Wang, and Wei Wang. 2014. Local search of communities in large graphs. In *SIGMOD*. 991–1002.

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

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

[17] Hossein Esfandiari, Silvio Lattanzi, and Vahab S. Mirrokni. 2018. Parallel and Streaming Algorithms for K-Core Decomposition. In *ICML*. 1396–1405.

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

[19] Edoardo Galimberti, Francesco Bonchi, and Francesco Gullo. 2017. Core Decomposition and Densest Subgraph in Multilayer Networks. In *CIKM*. 1807–1816.

[20] Edoardo Galimberti, Francesco Bonchi, Francesco Gullo, and Tommaso Lanciano. 2020. Core Decomposition in Multilayer Networks: Theory, Algorithms, and Applications. *ACM Trans. Knowl. Discov. Data* 14, 1 (2020), 11:1–11:40.

[21] Edoardo Galimberti, Martino Ciaperoni, Alain Barrat, Francesco Bonchi, Ciro Cattuto, and Francesco Gullo. 2019. Span-core Decomposition for Temporal Networks: Algorithms and Applications. *CoRR* abs/1910.03645 (2019). http://arxiv.org/abs/1910.03645

[22] Christos Giatsidis, Dimitrios M. Thilikos, and Michalis Vazirgiannis. 2011. D-cores: Measuring Collaboration of Directed Graphs Based on Degeneracy. In *ICDM*. 201–210.

[23] Jorge E. Hirsch. 2005. An index to quantify an individual's scientific research output. *Proc. Natl. Acad. Sci. U.S.A.* 102, 46 (2005), 16569–16572.

[24] Qiang-Sheng Hua, Yuliang Shi, Dongxiao Yu, Hai Jin, Jiguo Yu, Zhipeng Cai, Xiuzhen Cheng, and Hanhua Chen. 2020. Faster Parallel Core Maintenance Algorithms in Dynamic Graphs. *IEEE Trans. Parallel Distrib. Syst.* 31, 6 (2020), 1287–1300.

[25] Xin Huang, Laks V. S. Lakshmanan, and Jianliang Xu. 2019. *Community Search over Big Graphs*. Morgan & Claypool Publishers. https://doi.org/10.2200/S00928ED1V01Y201906DTM061

[26] Xin Huang, Wei Lu, and Laks V. S. Lakshmanan. 2016. Truss Decomposition of Probabilistic Graphs: Semantics and Algorithms. In *SIGMOD*. 77–90.

[27] Hai Jin, Na Wang, Dongxiao Yu, Qiang-Sheng Hua, Xuanhua Shi, and Xia Xie. 2018. Core Maintenance in Dynamic Graphs: A Parallel Approach Based on Matching. *IEEE Trans. Parallel Distrib. Syst.* 29, 11 (2018), 2416–2428.

[28] Wissam Khaouid, Marina Barsky, S. Venkatesh, and Alex Thomo. 2015. K-Core Decomposition of Large Networks on a Single PC. *Proc. VLDB Endow.* 9, 1 (2015), 13–23.

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

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

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

[32] Linyuan Lü, Tao Zhou, Qian-Ming Zhang, and H Eugene Stanley. 2016. The H-index of a network node and its relation to degree and coreness. *Nature communications* 7, 1 (2016), 1–7.

[33] Fragkiskos D. Malliaros, Christos Giatsidis, Apostolos N. Papadopoulos, and Michalis Vazirgiannis. 2020. The core decomposition of networks: theory, algorithms and applications. *VLDB J.* 29, 1 (2020), 61–92.

[34] Aritra Mandal and Mohammad Al Hasan. 2017. A distributed k-core decomposition algorithm on spark. In *2017 IEEE International Conference on Big Data, BigData 2017, Boston, MA, USA, December 11-14, 2017*. 976–981.

[35] S Thomas McCormick. 1983. Optimal approximation of sparse Hessians and its equivalence to a graph coloring problem. *Mathematical Programming* 26, 2 (1983), 153–171.

[36] Alberto Montresor, Francesco De Pellegrini, and Daniele Miorandi. 2013. Distributed k-Core Decomposition. *IEEE Trans. Parallel Distrib. Syst.* 24, 2 (2013), 288–300.

[37] Michael P. O'Brien and Blair D. Sullivan. 2014. Locally Estimating Core Numbers. In *ICDM*. 460–469.

[38] You Peng, Ying Zhang, Wenjie Zhang, Xuemin Lin, and Lu Qin. 2018. Efficient Probabilistic K-Core Computation on Uncertain Graphs. In *ICDE*. 1192–1203.

[39] Ahmet Erdem Sariyüce, Bugra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V. Çatalyürek. 2013. Streaming Algorithms for k-core Decomposition. *Proc. VLDB Endow.* 6, 6 (2013), 433–444.

[40] Ahmet Erdem Sariyüce, Bugra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V. Çatalyürek. 2016. Incremental k-core decomposition: algorithms and evaluation. *VLDB J.* 25, 3 (2016), 425–447.

[41] Ahmet Erdem Sariyüce, C. Seshadhri, and Ali Pinar. 2018. Local Algorithms for Hierarchical Dense Subgraph Discovery. *Proc. VLDB Endow.* 12, 1 (2018), 43–56.

[42] Mauro Sozio and Aristides Gionis. 2010. The community-search problem and how to plan a successful cocktail party. In *SIGKDD*. 939–948.

[43] Charalampos E. Tsourakakis, Francesco Bonchi, Aristides Gionis, Francesco Gullo, and Maria A. Tsiarli. 2013. Denser than the densest subgraph: extracting optimal quasi-cliques with quality guarantees. In *SIGKDD*. 104–112.

[44] Tatiana von Landesberger, Arjan Kuijper, Tobias Schreck, Jörn Kohlhammer, Jarke J. van Wijk, Jean-Daniel Fekete, and Dieter W. Fellner. 2011. Visual Analysis of Large Graphs: State-of-the-Art and Future Research Challenges. *Comput. Graph. Forum* 30, 6 (2011), 1719–1749.

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

[46] Na Wang, Dongxiao Yu, Hai Jin, Chen Qian, Xia Xie, and Qiang-Sheng Hua. 2017. Parallel Algorithm for Core Maintenance in Dynamic Graphs. In *ICDCS*. 2366–2371.

[47] Stanley Wasserman and Katherine Faust. 1994. *Social Network Analysis: Methods and Applications*. Cambridge University Press. https://doi.org/10.1017/CBO9780511815478

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

[49] Huanhuan Wu, James Cheng, Yi Lu, Yiping Ke, Yuzhen Huang, Da Yan, and Hejun Wu. 2015. Core decomposition in large temporal graphs. In *Big Data*. 649–658.

[50] Haohua Zhang, Hai Zhao, Wei Cai, Jie Liu, and Wanlei Zhou. 2010. Using the k-core decomposition to analyze the static structure of large-scale software systems. *J. Supercomput.* 53, 2 (2010), 352–369.

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

[52] Yi Zhou, Jingwei Xu, Zhenyu Guo, Mingyu Xiao, and Yan Jin. 2020. Enumerating Maximal k-Plexes with Worst-Case Time Guarantee. In *AAAI*. 2442–2449.