Hierarchical Core Maintenance on Large Dynamic Graphs

Zhe Lin
East China Normal University
linzhe@stu.ecnu.edu.cn

Fan Zhang
Guangzhou University
zhangf@gzhu.edu.cn

Xuemin Lin
University of New South Wales
East China Normal University
lxue@cse.unsw.edu.au

Wenjie Zhang
University of New South Wales
zhangw@cse.unsw.edu.au

Zhihong Tian
Guangzhou University
tianzhihong@gzhu.edu.cn

ABSTRACT

The model of k-core and its decomposition have been applied in various areas, such as social networks, the world wide web, and biology. A graph can be decomposed into an elegant k-core hierarchy to facilitate cohesive subgraph discovery and network analysis. As many real-life graphs are fast evolving, existing works proposed efficient algorithms to maintain the coreness value of every vertex against structure changes. However, the maintenance of the k-core hierarchy in existing studies is not complete because the connections among different k-cores in the hierarchy are not considered. In this paper, we study hierarchical core maintenance which is to compute the k-core hierarchy incrementally against graph dynamics. The problem is challenging because the change of hierarchy may be large and complex even for a slight graph update. In order to precisely locate the area affected by graph dynamics, we conduct in-depth analyses on the structural properties of the hierarchy, and propose well-designed local update techniques. Our algorithms significantly outperform the baselines on runtime by up to 3 orders of magnitude, as demonstrated on 10 real-world large graphs.

PVLDB Reference Format:
Zhe Lin, Fan Zhang, Xuemin Lin, Wenjie Zhang, and Zhihong Tian.
Hierarchical Core Maintenance on Large Dynamic Graphs. PVLDB, 14(5): 757-770, 2021.
doi:10.14778/3446095.3446099

1 INTRODUCTION

The structure modeling of complex networks has been widely studied in the form of graphs. Applications of graph analytics exist in various areas, and the mining of cohesive subgraphs is a fundamental graph problem. One of the most well-studied cohesive subgraph model is the k-core, defined as a maximal *connected* subgraph in which every vertex is connected to at least k other vertices in the same subgraph [42, 52]. For a fixed parameter k, there may be more than one k-core in the graph G, and we use the k-core set to denote the subgraph formed by all the (connected) k-cores in G. The coreness of a vertex is the largest k s.t. a k-core contains the vertex. A graph can be decomposed by k-core into an elegant hierarchical structure: for every integer k, (i) each k-core is contained by exactly one $(k-1)$-core, and (ii) for every integer k, the k-cores are disjoint. The k-core hierarchy can be represented by a tree, where each k-core S corresponds to a tree node containing the vertices with coreness k in S, and each tree edge represents a parent k-core containing its child k'-core with $k' > k$.

The k-core and its hierarchical decomposition have a wide spectrum of applications, e.g., discovering communities in the web [16] and social networks [19, 56, 65], modeling the dynamics of user engagement [41, 64], discovering molecular complexes in protein interaction networks [2], analyzing the underlying structure of the Internet and its functional consequences [4], and predicting structural collapse in mutualistic ecosystems [44]. The hierarchical structure is effective to locate communities of a network and explore the insights of network phenomena [12].

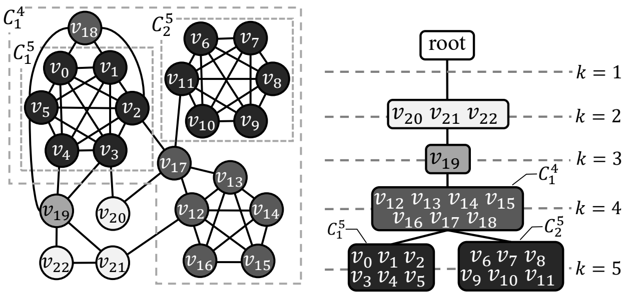
In the detection of cohesive subgraphs, a recent work computes the (connected) k-core C^* with the largest density (average degree) for any k value in the hierarchy of core decomposition, which is the state-of-the-art approximate solution to find the densest subgraph [11]. This algorithm is much more efficient than other approaches. The resulting C^* has a 0.5-approximation guarantee and often a higher density than other approximations. It is validated that finding C^* can also help the computation of maximum clique and size-constraint k-core. For the study of user engagement, the coreness of a vertex is regarded as the “best practice” to capture its engagement level [41]. It is validated that the average engagement (e.g., the number of check-ins) of the vertices with a same coreness is usually in a positive correlation with the value of their coreness [35]. However, our experimental results find that the engagement evaluation of a vertex can be more accurate by considering both its coreness and its position in the hierarchy of core decomposition.

In real-life, many graphs are highly dynamic, e.g., the users in a social network may add new friends or remove existing friend relations, new links are constantly established in the web due to the creation of new pages. Consequently, there are numerous studies on dynamic graphs, e.g., [18, 69], where vertices/edges will be inserted/removed dynamically. Nevertheless, the existing works of core decomposition on dynamic graphs focus on maintaining the coreness of each vertex [49, 60, 66], while the maintenance of the connections among k-cores in the hierarchy are not considered . As the structural information is critical for core decomposition (e.g., the k-core is defined on connected subgraphs), in this paper, we study the problem of hierarchical core maintenance, which is to update

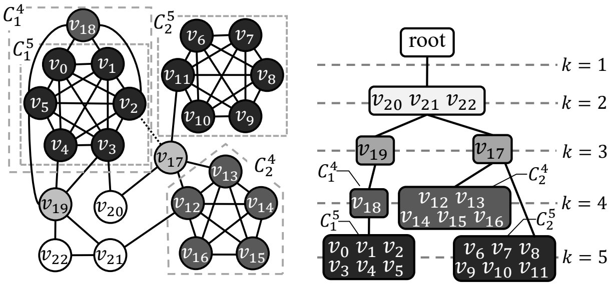
---

[*] Zhe Lin and Fan Zhang are the joint first authors.

Proceedings of the VLDB Endowment, Vol. 14, No. 5 ISSN 2150-8097.
doi:10.14778/3446095.3446099



(a) graph (b) k-core hierarchy



(c) remove edge (v_2, v_{17}) (d) updated k-core hierarchy

Figure 1: Hierarchical Core Decomposition

the k-core hierarchy incrementally against edge insertion/removal. The insertion (resp. removal) of vertices can be simulated as a sequence of edge insertions (resp. edge removals) [66].

Example 1. *Figure 1(a) illustrates a graph G with 23 vertices and their connections, where the color depth of a vertex represents the coreness of this vertex. The whole graph is a 2-core, and the 3-core is induced by $V(G) \setminus \{v_{20}, v_{21}, v_{22}\}$. The 4-core is induced by the 3-core minus v_{19}. So, the coreness of v_{19} is 3. The core decomposition of G iteratively removes a vertex with the smallest degree in current G s.t. the k-cores with different k values are retrieved. The 5-cores (C_1^5 and C_2^5) are circled by dotted lines.*

The k-core hierarchy $T(G)$ is shown in Figure 1(b) which contains both the coreness of each vertex and the connections among different k-cores. For instance, the coreness of v_{17} is 4 as it is on the 4^{th} layer. Let n_1 denote the node that includes v_{17}, the 4-core containing v_{17} is induced by the subtree rooted at n_1, which contains two 5-cores.

If we remove the edge (v_2, v_{17}) from G, the coreness value for each vertex and the k-cores will be updated as shown in Figure 1(c). As the coreness of v_{17} decreases to 3, it is moved to the 3^{rd} layer in Figure 1(d). Then, the previous 4-core (C^4) splits to two 4-cores C_1^4 and C_2^4. Accordingly, the node at the 4^{th} layer splits to two nodes as shown in the figure. The tree edges of $T(G)$ are adjusted based on the containment property of different k-cores.

The coreness of each vertex in a graph can be computed in linear time by core decomposition, which iteratively removes a vertex with the smallest degree in the remaining graph [3]. The state-of-the-art solution for updating the vertex coreness is proposed in [66], based on the vertex deletion order in core decomposition. However, as the algorithm is only designed for updating coreness, we have to traverse the graph with the updated coreness to find a (connected) k-core. It means that, for the maintenance of the k-core hierarchy on dynamic graphs, the existing solutions have to rebuild the k-core hierarchy from scratch by executing its construction algorithm which is costly. The state-of-the-art algorithm (named LCPS) constructs the k-core hierarchy on static graphs with a time complexity of $O(m)$ [42], if buckets are used to maintain the search priority [51]. It sequentially pushes a vertex and its neighbors (unvisited) into queues according to a priority function s.t. the subtree containing the vertex is traversed and built. As the search priority in LCPS can be either partially bottom-up or partially top-down, given a set of inserted/removed edges, we have to execute LCPS from scratch on G or a part of G which is cost-prohibitive. Thus, in this paper, we aim to incrementally update the k-core hierarchy through precisely locating the structure of the hierarchy affected by the graph updates.

The problem of hierarchical core maintenance is challenging because the hierarchy may change a lot even for a slight graph update, and the connectivity changes among different k-cores are non-trivial, as illustrated in Example 1. In order to capture the effect of an inserted/removed edge towards the k-core hierarchy T, we conduct in-depth analyses on the structural connections of the k-cores in the hierarchy. A series of theorems are proposed to fast identify the unchanged structure in T, and facilitate efficient update operations, e.g., node mergence, node split, and adjusting the parent-child relations. Our focus is to propose the maintenance solution for instant update, i.e., the case of one inserted/removed edge. The algorithms are extended to address the (one-time) update for multiple inserted/removed edges. Several well-designed local update techniques are proposed s.t. the k-core hierarchy can be maintained efficiently on large dynamic graphs.

Contributions. The principal contributions are as follows.

- To the best of our knowledge, this is the first work to study hierarchical core decomposition on dynamic graphs.
- In-depth analyses are conducted to explore the structural change of the k-core hierarchy. A series of theorems are presented to tackle the effect of edge insertion/deletion on the k-core hierarchy.
- Efficient algorithms are proposed for hierarchical core maintenance against the insertion/removal of one edge, with effective local update techniques. The algorithms are extended to handle multiple inserted/removed edges in a batch.
- Extensive experiments are conducted on 10 real-world networks with up to billions of edges. Our algorithms outperform the baselines on runtime by up to 3 orders of magnitude. The algorithms are shown effective in maintaining cohesive subgraphs and evaluating user engagement.

2 PRELIMINARIES

We consider an unweighted and undirected graph $G = (V, E)$, with $n = |V|$ vertices and $m = |E|$ edges (assume $m > n$). A graph G' is the subgraph of G, denoted by $G' \subseteq G$, if $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$. The notations are summarized in Table 1.

Table 1: Summary of Notations (When the context is clear, we abbreviate the notations, e.g., using $N(v)$ instead of $N(v,G)$)

Notation	Definition
$G=(V,E)$	an undirected and unweighted graph
$V(G), E(G)$	the vertex/edge set of G
$N(v,G)$	the neighbor set of vertex v in G
$C_i^k; C^k(v,G)$	a k-core; the k-core includes v in G
$core(v,G)$	the coreness of v in G
$C(v,G)$	$C^{core(v)}(v)$ on G
$T(G)$	k-core hierarchy of graph G
T	a k-core hierarchy
n_1	a tree node (on T)
$V(n_1)$	the vertex set of n_1
$L_k(T)$	the k^{th} layer of T
$core(n_1,G)$	$core(v,G)$ for any $v \in V(n_1)$
$T'(n_1)$	the subtree rooted at n_1
$G[n_1]$	the subgraph of G induced by the vertices in $T'(n_1)$
G_0	the original graph
E'	the edge set inserting into/removing from G_0
G^*	the graph with inserted/removed edges
V^*	$\{v \in V(G) \mid core(v,G_0) \neq core(v,G^*)\}$
T_0, T^*	the k-core hierarchy of G_0 and G^*, respectively
$node(v,T)$	the tree node of T containing vertex v
$P(n_1)$	the parent node of n_1 in $T(G)$
$cn(n_1,v)$	a child node n_c of n_1's with $v \in T'(n_c,T)$
$cn(n_1,n_2)$	$cn(n_1,v)$ for any $v \in V(n_2,T)$

2.1 Core Maintenance

Definition 1 (k-core [42, 52]). *Given a graph G and an integer k, a subgraph S is a k-core of G, if (i) each vertex $v \in S$ has at least k neighbors in S, i.e., $|N(v,S)| \geq k$; (ii) S is connected; and (iii) S is maximal, i.e., any supergraph of S is not a k-core except S itself. Let C_i^k denote the i^{th} k-core of G for a given k.*

Given a fixed integer k, we use the k-core set to denote the subgraph that containing every (connected) k-core.

Definition 2 (k-core set). *Given a graph G and an integer k, the k-core set of G is the subgraph formed by all the (connected) k-cores in G, i.e., $\cup_{i \in N^+}\{C_i^k\}$.*

If $k' \geq k$, the k'-core set is always a subgraph of (i.e., contained by) the k-core set. Each vertex in G has a fixed coreness value [3].

Definition 3 (coreness). *Given a graph G, the coreness of a vertex $v \in V(G)$, denoted by $core(v)$, is the largest k such that v is in the k-core, i.e. $core(v) = \max_{v \in C^k}\{k\}$.*

For any vertex $v \in V(G)$, we use $C^k(v)$ to denote the k-core containing the vertex v. Let $C(v)$ represent $C^k(v)$ with $k = core(v)$.

Definition 4 (core decomposition). *Given a graph G, core decomposition is to compute the coreness $core(v)$ for every vertex $v \in V(G)$.*

The algorithm of core decomposition recursively removes a vertex with the smallest degree in the remaining graph, with a time complexity of $O(m)$ [3].

Definition 5 (core maintenance). *Given a graph G, if the edges in E' are inserted into (resp. removed from) G, core maintenance is to update the corenesses of all the vertices, after the graph (V,E) evolves to $(V, E+E')$ (resp. $(V, E-E')$).*

The algorithm of core maintenance utilizes the ordering of vertex removal in core decomposition to fast update the corenesses of the vertices affected by edge insertion/deletion [66].

2.2 Hierarchical Core Maintenance

According to the definition of k-core, we can get the following two properties for every integer k:

- *Containment.* Each k-core is contained by exactly one $(k-1)$-core.
- *Disjointness.* $C_1^k \cap C_2^k = \emptyset$, for any two different k-cores C_1^k and C_2^k.

Given a graph G, the k-core hierarchy of G can be represented by a tree, where each k-core of G is induced by a subtree of $T(G)$.

Definition 6 (k-core hierarchy). *Given a graph G, the k-core hierarchy, denoted as $T(G)$, is a tree structure containing all the k-cores and their connections, for every k value:*

- *Tree Node.* For each k-core C_i^k in G, there is a uniquely associated tree node n_1 located at the k^{th} layer of T, if there is at least one vertex in C_i^k with coreness equals to k. The node n_1 contains all the vertices in C_i^k with coreness k, i.e. $V(n_1) = \{v \mid v \in C_i^k \wedge core(v) = k\}$.
- *Tree Edge.* For a k_1-core $C_i^{k_1}$ associated with tree node n_1, and a k_2-core $C_j^{k_2}$ associated with n_2 in G, the tree node n_1 is the parent node of n_2 iff (i) $k_1 < k_2$; (ii) $C_j^{k_2} \subset C_i^{k_1}$; and (iii) for any k'-core with $k_1 < k' < k_2$, the associated tree node is not the parent of n_2.
- *Root.* The isolated vertices are recorded in the root node of $T(G)$. We create a tree edge between the root and each tree node associated with a connected component of G (i.e., each 1^{st} layer node).

The k-core hierarchy can be constructed in $O(m)$ time by LCPS algorithm (level component priority search) [42, 51]. It sequentially pushes a vertex and its neighbors (unvisited) into queues s.t. the subtree containing the vertex is traversed and built.

To avoid ambiguity, we use *vertex* to indicate the vertex in $V(G)$ and *(tree) node* to indicate the node in $T(G)$.

Given a graph G, its k-core hierarchy $T = T(G)$, and a node $v \in V(G)$. We use $node(v,T) = n_1$ to denote the tree node n_1 containing vertex v. For the node n_1, let $T'(n_1)$ denote the subtree rooted at the n_1 and $G[n_1]$ denote the subgraph induced by the vertices in $T'(n_1)$. We have $G[n_1] = C(v,G)$. Let $L_k(T)$ denote the k^{th} layer of $T(G)$.

Problem Definition. Given a graph G, and the edges set E' inserting to (resp. removing from) G. Let G_0 denote the original graph, i.e., $G_0 = G$. Let G^* denote the changed graph, i.e., $G^* = (V, E+E')$ (resp. $G^* = (V, E-E')$). Hierarchical core maintenance is to update the k-core hierarchy from $T(G_0)$ to $T(G^*)$.

If a vertex v is inserted to the graph, we first record v in the root node, and then maintain the k-core hierarchy by inserting every edge incident to v. If a vertex u is removed from the graph, we maintain the hierarchy by deleting each edge incident to v. As the update of vertices can be processed by the update of their incident edges, we focus on edge insertion/deletion in this paper.

Let $T_0 = T(G_0)$, $T^* = T(G^*)$, and T denote current k-core hierarchy under the maintenance process. In the algorithms, we divide the maintenance process into several stages. After processing each stage, we use $T_1, T_2, ..., T_n$ to record current T.

Given a node n_i in T_i, $P(n_i)$ denotes the parent node of n_i, and $T'(n_i)$ denotes the subtree rooted at n_i in T_i. Given an integer j, if a node n_j in T_j satisfies $T'(n_j) = T'(n_i)$ (resp. $V(n_j) = V(n_i)$), we say $T'(n_i)$ (resp. n_i) *keeps the same in* T_j. If $T'(n_i)$ keeps the same in T_j, n_i also keeps the same in T_j.

3 EDGE INSERTION

In this section, we first maintain the k-core hierarchy against one inserted edge, and then address a batch of inserted edges.

3.1 Insertion Analysis

Let (x_1, x_2) denote the edge to be inserted into G_0, where $(x_1, x_2) \notin E(G_0)$. W.l.o.g, we suppose $K = core(x_1, G_0) \le core(x_2, G_0)$.

In order to better present our algorithm, we first analyze the effect of inserting one edge.

Coreness Update. After the insertion of (x_1, x_2), we adopt the state-of-the-art algorithm for core maintenance [66] to update the corenesses of all the affected vertices. Let V^* denote the set of vertices with coreness changed after the insertion of (x_1, x_2). According to existing study of maintaining coreness, there exist some key rules for the insertion of one edge [33, 49].

- For each vertex $v \in V^*$, we have $core(v, G_0) = K$ and $core(v, G^*) = K + 1$.
- If $core(x_1, G_0) < core(x_2, G_0)$, we have $V^* \subseteq V(C(x_1))$, the subgraph induced by V^* on G_0 is connected, and $x_1 \in V^*$.
- If $core(x_1, G_0) = core(x_2, G_0)$, we have $V^* \subseteq \{V(C(x_1)) \cup V(C(x_2))\}$. The subgraph induced by V^* on G_0 either is connected, or consists of two connected components that one contains x_1 and the other contains x_2.
- The induced subgraph of V^* in G^* is connected.

Hierarchy Analysis. For the update of k-core hierarchy with inserting (x_1, x_2), we discuss the following cases for each k-core C_i^k in the original graph G_0. Note that $K = core(x_1) \le core(x_2)$.

(i) $k > K + 1$. For every vertex v with $core(v, G_0) > K + 1$, we have $core(v, G^*) = core(v, G_0)$. C_i^k keeps the same after the insertion, as C_i^k does not contain x_1, x_2, or any vertex in V^*.

(ii) $k \le K$. (a) If C_i^k contains either x_1 or x_2. W.l.o.g, suppose we have $x_1 \in C_i^k$, the insertion of (x_1, x_2) will connect (merge) C_i^k and $C^k(x_2)$. (b) Besides, if $k = K$, the coreness of each vertex in V^* increases to $K + 1$ from K. C_i^k may lose some vertices(i.e., in V^*) and we will discuss this case in details later.

(iii) $k = K + 1$. The vertices in V^* may connect to C_i^k on G^*. We will discuss this case later too.

Algorithm 1: **InsertOne**

```
   Input  : a graph $G_0$, the $k$-core hierarchy $T_0$, an edge
            $(x_1, x_2) \notin E(G_0)$
   Output : $T^*$
1  $T \leftarrow T_0$; $G \leftarrow G_0$; $K \leftarrow core(x_1)$ (suppose $core(x_1) \le core(x_2)$);
2  $V^* \leftarrow$ vertices with coreness changed by inserting $(x_1, x_2)$ to $G$;
3  $n_1 \leftarrow node(x_1)$; $n_2 \leftarrow node(x_2)$;
4  while $n_1 \ne n_2$ do
5      swap $n_1$ and $n_2$ if $core(n_1) > core(n_2)$;
6      $p_1 \leftarrow P(n_1)$; $p_2 \leftarrow P(n_2)$;
7      if $core(n_1) = core(n_2)$ then
8          $n_0 \leftarrow$ merge $n_1$ and $n_2$ in $T$;
9          $P(n_0) \leftarrow p_1$ or $p_2$ whose coreness is larger;
10         $n_1 \leftarrow p_1$; $n_2 \leftarrow p_2$;
11     else
12         $P(n_2) \leftarrow n_1$ if $core(n_1) > core(p_2)$;
13         $n_2 \leftarrow p_2$;
14 $T_1 \leftarrow T$; $n' \leftarrow node(V^*)$ of $T$;
15 create a node $n^+$ on $L_{K+1}$ in $T$ as a child of $n'$;
16 move $v$ to $V(n^+)$ from $V(n')$ for each $v \in V^*$;
17 $NC = \{cn(n', u, T) \mid u \in N(V^*, G^*)\}$; $T_2 \leftarrow T$;
18 for each $n_c \in NC$ do
19     if $core(n_c, G^*) = K + 1$ then
20         merge $n_c$ into $n^+$;
21     else
22         $P(n_c, T) \leftarrow n^+$;
23 if $V(n') = \emptyset$ then
24     $P(n_0) \leftarrow P(n')$ for each child $n_0$ of $n'$;
25     remove $n'$ from $T$;
26 return $T$ (i.e., $T^*$)
```

3.2 Merge Ancestors of $node(x_1)$ and $node(x_2)$

As shown in the above subsection, the hierarchy of k-cores keeps the same in case (i), i.e., when $k > K + 1$. For case (ii), we show how to merge the k-cores in this subsection. We leave the techniques for case (iii) in next subsection.

According to the definition of k-core hierarchy, the ancestors of $node(x_1, T_0)$ and $node(x_2, T_0)$ at the same layer of T_0 will be merged into one tree node of T^*. After merging the ancestor nodes at the same layer, the tree edges (parent-child relations) incident to them should be adjusted accordingly. For the nodes not on the branches containing x_1 or x_2, Theorem 1 proves that the associated k-cores of these nodes keep the same for the insertion of (x_1, x_2).

Theorem 1. *For any tree node $n_0 \in T_0$ satisfying $G_0[n_0] \cap \{C(x_1) \cup C(x_2)\} = \emptyset$, we have $T'(n_0)$ keeps the same in T^*.*

Proof. Let $k_0 = core(n_0)$. As $G_0[n_0] \cap \{C(x_1) \cup C(x_2)\} = \emptyset$ and $V^* \subseteq V(C(x_1) \cup C(x_2))$, in core decomposition of G^*, the vertices in all ancestors of n_0 will still be deleted when we compute the k_0-core set of G^*. Thus, $T'(n_0)$ keeps the same in T^*. □

Line 1-13 of Algorithm 1 shows the pseudo-code to merge the ancestors of $node(x_1)$ and $node(x_2)$. Suppose $core(x_1) \le core(x_2)$, the coreness value of $core(x_1)$ is recorded by K at Line 1. After the insertion of (x_1, x_2), we update the coreness of each vertex at Line

2 by the state-of-the-art algorithm in [66]. Let n_1 and n_2 denote the two nodes under processing, which are initialized by $node(x_1)$ and $node(x_2)$, respectively (Line 3).

We merge the tree nodes in a bottom-up manner from n_1 and n_2 in T until there is no further mergence, i.e., $n_1 = n_2$ (Line 4-13). In each iteration, we use Line 5 to ensure $core(n_1) \leq core(n_2)$, and record the parent nodes of n_1 and n_2 (Line 6). (i) When $core(n_1) = core(n_2)$, we merge n_1 and n_2 to n_0 where the child relations inherit (Line 8). Then, we adjust the parent relation of n_0 based on the definition of k-core hierarchy (Line 9). (ii) When $core(n_1) < core(n_2)$, we only need to adjust the parent relation of n_2 (Line 12). The next nodes to process are set accordingly (Line 10 or 13).

After the mergence (Line 1-13), we get an intermediate k-core hierarchy T_1 where the upper part (each L_k with $k < K$) has been maintained correctly.

Example 2. *A graph G_0 is shown in Figure 2(a) and its k-core hierarchy T_0 is depicted in Figure 2(b). If an edge (v_0, v_5) is inserted to G_0, after running Line 1-14 of Algorithm 1, we will retrieve an intermediate T_1 as shown in Figure 2(c). In the first iteration, as $core(v_5) < core(v_0)$, we have $n_1 = node(v_5)$ and $n_2 = node(v_0)$ after running Line 5. Since $core(n_1) < core(P(n_2))$ at Line 12, we just set $n_2 = P(n_2)$. In the second iteration, as $core(n_1) > core(P(n_2))$, we set $node(v_5)$ as the parent node of $node(v_{15})$. Iteratively, we merge the ancestors of $node(v_5)$ and $node(v_0)$, and retrieve T_1.*

3.3 Adjust the Subtree under L_K.

After merging the ancestors of $node(x_1)$ and $node(x_2)$, we need to adjust some tree nodes in $L_K(T_1) \cup L_{K+1}(T_1)$ and the associated edges (parent-child relations). We first show that there is a node n'_1 in T_1 containing V^*, and only the subtree rooted at the node n'_1 in T_1 should be updated in the maintenance.

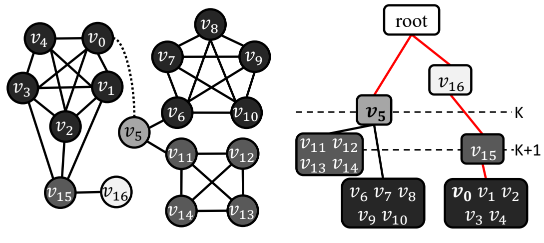
Theorem 2. *(i) There is a node $n'_1 \in T_1$ satisfying $V^* \subseteq V(n'_1)$. (ii) For any node $n_0 \in T_1$ with $G^*[n_0] \cap G^*[n'_1] = \emptyset$, $T'(n_0)$ keeps the same in T^*.*

Proof. (i) When $core(x_1, G_0) < core(x_2, G_0)$, we have $n'_1 = node(x_1, T_1)$ and $V^* \subseteq V(n'_1)$. When $core(x_1, G_0) = core(x_2, G_0)$, since $node(x_1, T_0)$ and $node(x_2, T_0)$ are merged in T_1, we have $V(n'_1) = V(node(x_1, T_0)) \cup V(node(x_2, T_0))$, and thus $V^* \subseteq V(n'_1)$. (ii) Similar to Theorem 1, for any node n_0 with $G^*[n_0] \cap G^*[n'_1] = \emptyset$, core decomposition on $G^*[n_0]$ is the same to that on $G[n_0]$. Thus, $T'(n_0)$ keeps the same in T^*. □

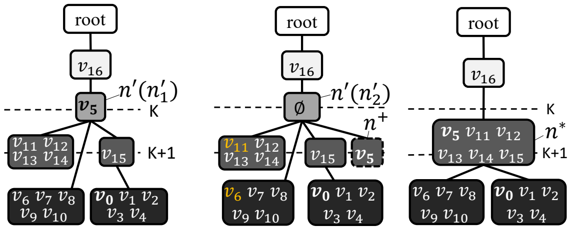
Theorem 3. *There is a node $n^* \in T^*$ satisfying $V^* \subseteq V(n^*)$.*

Proof. The proof is straightforward as $core(v, G^*) = K + 1$ for each $v \in V^*$, and the induced subgraph of V^* in G^* is connected. □

Then, we compute the next intermediate hierarchy T_2 from Line 14 of Algorithm 1. According to Theorem 2, let n' denote the node in T equals to n'_1 when recording T_1 (Line 14), and n'_2 denote the node in T_2 equals to n' when recording T_2 (Line 17). At Line 15, we create a node n^+ in $L_{K+1}(T)$ as the child node of n', to process the node adjustment. The vertices in V^* are moved to $V(n^+)$, as their coreness increases to $K + 1$ from K (Line 16). Now we get T_2 where each vertex is in the correct layer and the first K layers are maintained except n'_2 if it becomes empty. If n'_2 is empty, we address it later in Line 23-25 for fewer operations.



(a) $G_0 + (v_0, v_5)$ (b) T_0



(c) T_1 (d) T_2 (e) T^*

Figure 2: Insert (v_0, v_5) to G_0

Remaining Nodes to Update. In order to find out the operations required for completing the maintenance, we introduce the notation $cn(n_0, v_0)$. Given a node n_0 and a vertex v_0, let $cn(n_0, v_0)$ denote the node n_c satisfying $P(n_c) = n_0$ and $T'(n_c)$ contains v_0. For example in Figure 2(d), $cn(n', v_0) = node(v_{15}, T_2)$.

Let $N(V^*, G^*) = \cup_{v \in V^*} N(v, G^*)$. After running Line 17 (recording T_2) of Algorithm 1, the set of candidate nodes to update in T_2 is **NC** $= \{cn(n'_2, u) \mid u \in N(V^*, G^*)\}$. For each node n_c in NC (note that $n_c \in T_2$), there is at least one vertex u in $T'(n_c)$ where u is the neighbor of a vertex in V^*.

Theorem 4. *For each $n_c \in NC$, $G^*[n_c] \subseteq G^*[n^*]$ holds.*

Proof. For each $n_c \in NC$, according to NC's definition, we have the following two properties: (i) $core(n_c) \geq core(n^+) = K + 1$; (ii) $\exists v_1 \in n_c, v_2 \in n^+$(i.e., V^*) satisfying $(v_1, v_2) \in E(G^*)$. According to the definition of k-core hierarchy, the vertices of n_c and n^+ are in one node of T^* when $core(n_c) = K + 1$ (Line 20 of Algorithm 1), or n^+ will be the ancestor of n_c when $core(n_c) > K + 1$ (Line 22). Thus, $G^*[n_c] \subseteq G^*[n^*]$ holds as $V^* = V(n^+) \subseteq V(n^*)$. □

For each node $n_c \in NC$, according to Theorem 4, we can adjust it easily in Line 20 or 22. When Algorithm 1 is returned, we have $T^* = T$ and n^+ of T is exactly n^* of T^* in Theorem 3. For conciseness, we defer the computation of $cn(n', u)$ to Algorithm 5 at Section 4.2. For the remaining nodes not in NC, the following theorem holds.

Theorem 5. *For any node $n_0 \in T'(n'_2)$ and $n_0 \notin NC$, $T'(n_0)$ keeps the same in T^*.*

Proof. For each n_0 mentioned above, if $core(n_0) > K+1$, $T'(n_0)$ keeps the same according to the point (i) in the hierarchy analysis of Section 3.1; if $core(n_0) = K + 1$, the definition of NC implies that there is no vertex in n_0 which is the neighbor of V^*, and $T'(n_0)$ keeps the same. □

```
Algorithm 2: InsertX
   Input  : a graph $G_0$, the $k$-core hierarchy $T_0$, an edge set
            $E' \nsubseteq E(G_0)$
   Output : $T^*$, i.e., the updated $T_0$
1  $V^* \leftarrow \emptyset$; $C \leftarrow \emptyset$; $G^* \leftarrow G_0$; $T \leftarrow T_0$;
2  for each $e \in E'$ do
3      $V' \leftarrow$ vertices with coreness changed by inserting $e$ to $G^*$;
4      $\mathbb{N} \leftarrow$ the set of $node(v)$ in $T$ for each $v \in V'$;
5      $n' \leftarrow$ any node from $\mathbb{N}$;
6      create $n^*$ on $(core(n') + 1)^{th}$ layer in $T$ as a child node of $n'$;
7      $C \leftarrow C \cup \{(n^*, n_0)\}$ for each $n_0 \in \mathbb{N}$;
8      move each $v \in V'$ to $V(n^*)$; remove empty nodes in $T$;
9      $G_0 \leftarrow G_0 + \{e\}$; $V^* \leftarrow V^* \cup V'$;
10 $T_1 \leftarrow T$;
11 for each $(u, v) \in E'$ do
12     $C \leftarrow C \cup (node(u, T), node(v, T))$;
13 for each $v \in V^*$ do
14     for each $u \in N(v, G^*)$ with $core(u, G^*) > core(v)$ do
15         $C \leftarrow C \cup (node(u, T), node(v, T))$;
16 for each integer $K$ from $k_{max}$ to 0 do
17     $n_0 \leftarrow$ an unvisited node in a node pair of $C$ with $core(n_0) = K$;
18     $\mathbb{N}_1 \leftarrow \{n_0\}$; $\mathbb{N}_2 \leftarrow \emptyset$;
19     while there is an unvisited node $n_1$ in $\mathbb{N}_1$ do
20         $\mathbb{N}_2 \leftarrow \mathbb{N}_2 \cup \{P(n_1)\}$; $n_1 \leftarrow$ visited;
21         for each node $n_2$ with $(n_1, n_2) \in C$ do
22             if $core(n_2) = K$ then
23                 $\mathbb{N}_1 \leftarrow \mathbb{N}_1 \cup \{n_2\}$;
24             else
25                 $\mathbb{N}_2 \leftarrow \mathbb{N}_2 \cup \{n_2\}$;
26     $n' \leftarrow$ a node in $\mathbb{N}_2$ with the largest coreness;
27     $C \leftarrow C \cup (n', n_2)$ for each $n_2 \in \mathbb{N}_2$;
28     merge $n_1$ into $n_0$ for each $n_1 \in \mathbb{N}_1$;
29     $P(n_0) \leftarrow n'$;
30 return $T$, i.e., $T^*$;
```

As the hierarchy of k-cores (the subtrees) with $k > K + 1$ keeps the same in G_0 and G^*, only these nodes on the $(K+1)^{th}$ layer need to be merged. According to Theorem 4, we merge them with n^+ (Line 20). The other nodes (not at L_{K+1}) in NC need to correct their parent relations (Line 22). After all the updates in Line 18-22, if $V(n')$ is empty, we set the parent of n^+ to $P(n')$, and remove n' from T (Line 23-24). Then, the maintenance is completed.

Example 3. *For the graph in Figure 2(a), after running Line 1-13 of Algorithm 1, we get* T_1 *in Figure 2(c). Then, as* $V^* = \{v_5\}$*, we have* $n' = node(v_5)$ *by Line 14. At Line 15,* n^+ *is created as a child node of* n' *and collects* v_5 *from* n'*.* T_2 *is shown in Figure 2(d). Next, at Line 18-22, we select the nodes containing the neighbors of* V^**, i.e.,* $node(v_0)$*,* $node(v_6)$*, and* $node(v_{11})$*, and mark their ancestors with the smallest coreness while not less than* $K + 1 = 4$*, i.e.,* $node(v_{15})$*,* $node(v_6)$*, and* $node(v_{11})$*. Then, for above nodes which are at* L_{K+1}*, we merge them with* n^+*, and change the parent of* $node(v_6)$ *to* n^+*. After the processing when* n' *is empty (Line 24-25), we get* T^* *in Figure 2(e).*

Correctness. The correctness of Algorithm 1 is guaranteed by the theorems in this section. According to Theorem 1 and 2, after moving V^* to the created node n^+ and removing n' if it is empty, all nodes in T_2 are already maintained except those in $T'(n'_2)$. For all nodes in $T'(n'_2)$, by Theorem 4, the nodes in NC are maintained correctly in Line 19-22; by Theorem 5, the other nodes (not in NC) keep the same in T^*.

Complexity. The space complexity of Algorithm. 1 is $O(|V| + |E|)$, as $|T_0| \leq |V|$. The time cost of it is the sum of three parts as follows. (i) In Line 1-13, we can get the complexity analysis of maintaining coreness (Line 2) from [66], and it runs in $O(\log \max\{|O_K|, |O_{K+1}|\} \times \sum_{v \in V^+} |N(v, G^*)|)$, where O_K is the set of vertices with coreness K, and V^+ is the subset of O_K, which is the vertex candidate set whose coreness may increase after the insertion.Then it takes $O(k_{max})$ to merge the ancestors of $node(x_1)$ and $node(x_2)$, where k_{max} is the height of T_0. (ii) In Line 14-16, the running time of moving the vertices in V^* is $O(|O_K|)$, as $V(node(x_1)) \cup V(node(x_2)) \in O_K$. (iii) In Line 17-25, adjusting the subtree rooted at n' takes $O(|T'(node(x_1))| + |T'(node(x_2))|)$ as there are at most $|T'(n')|$ nodes to adjust their parent relations, and $T'(n') \subset T'(node(x_1)) \cup T'(node(x_2))$. Thus, the time complexity of Algorithm 1 is $O((\log \max\{|O_K|, |O_{K+1}|\} \times \sum_{v \in V^+} |N(v, G^*)|) + k_{max} + |V^*| + |T'(node(x_1))| + |T'(node(x_2))|)$.

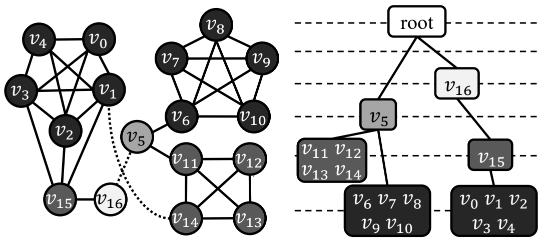
3.4 Insertion of x Edges

In this section, the k-core hierarchy is updated in a batch-processing manner, i.e., update once for the insertion of multiple edges. Compared with the update for one inserted edge, it is more complex to update T_0 with the insertion of x edges: (1) the coreness of each vertex may increase by more than 1; and (2) the affected area in T_0 would be larger.

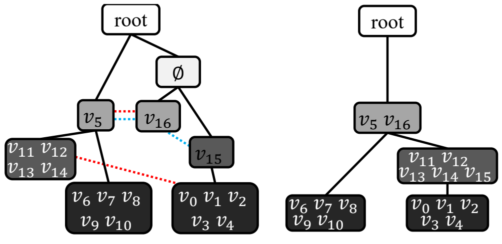
Let E' denote the edge set to be inserted to G_0, where $E' \nsubseteq E(G_0)$. Accordingly, we use G^* to denote $G_0 + E'$.

Algorithm 2 shows the maintenance against the insertion of E'. In order to avoid unnecessary cost (e.g., duplicate visit of some nodes), we defer the adjustment (Line 11-13, 17-25 of Algorithm 1) of the nodes in T_0 to the last part of the algorithm (Line 16-29 of Algorithm 2), and use the candidate set C to store all the node pairs that need to be adjusted (Line 7, 11-15 and 27), i.e., the nodes will either be merged at the same layer, or have their parent-child relations to be updated. Essentially, the mergence of ancestors (corresponding to Section 3.2) and the subtree adjustment (corresponding to Section 3.3) are combined by exploring the set C once in a bottom-up manner on the tree (Line 16-29). When $|E'| = 1$, Algorithm 2 is actually the same to Algorithm 1.

In details, we insert each edge $e \in E'$ one by one, and generate V' which contains all the vertices with coreness changed with inserting e, by coreness maintenance [66] (Line 2-3). Similar to Line 14-16 of Algorithm 2, we create a node n^* as a child node of n' in T, where n' is a node containing at least one vertex in V' (Line 4-6). Note that the vertices in V' may locate at different nodes, as T has not been fully updated in former iterations. Let $\mathbb{N}$ be a node set where each node contains at least one vertex in V' (Line 4). According to Theorem 3, the nodes in $\mathbb{N}$ will be merged into one node on T^* at last, we can set n' as any node from $\mathbb{N}$ in Line 5. Thus, the node pair (n^*, n_0) for each $n_0 \in \mathbb{N}$ is added to the candidate set C in Line



(a) $G_0 + (v_1, v_{14}) + (v_5, v_{16})$ (b) T_0



(c) T_1 (d) T^*

Figure 3: Insert (v_1, v_{14}) and (v_5, v_{16}) to G_0

7 for later update. In Line 8, we move each vertex v in V' to the correct layer, i.e., from $V(node(v))$ to $V(n^*)$. The empty nodes are removed and the related tree edges are adjusted. Line 9 updates G_0 and V^*. As the vertices in V' may be moved in later iterations, we only mark the vertices to move and the nodes containing them, in order to move them by one time after running Line 2-9.

We get T_1 till now, in which each vertex is in the correct layer. According to Algorithm 1, an edge $(u, v) \in E(G^*)$ may incur the update of the tree structure, where (u, v) must meet either (i) $(u, v) \in E'$ or (ii) $v \in V^* \wedge u \in N(v, G^*) \wedge core(u, G^*) > core(v)$. At Line 11-12, we add all the candidate edges to C for case (i); The candidate edges from case (ii) are added to C at Line 13-15.

Example 4. *For the graph G_0 in Figure 3(a), its k-core hierarchy is shown in Figure 3(b). We first insert the edge (v_1, v_{14}), and find $V' = \emptyset$ s.t. no update is triggered in Line 3-9. Next, we insert another edge (v_5, v_{16}), and find $V' = \{v_{16}\}$. At Line 6, a tree node n^* is created in the layer $L_{core(v_{16})+1} = L_2$, as a child node of $n' = node(v_{16})$, and v_{16} is moved from n' to n^*, as shown in Figure 3(c). The node pair (n^*, n') is added to C at Line 7. Now we focus on the intermediate tree T_1, for the candidates in C from E' (Line 11-12), we add $(node(v_5), node(v_{16}))$ and $(node(v_1), node(v_{14}))$ to C, as marked by red dashed lines in Figure 3(c). The other candidates in C are from the neighbors of each vertex in $V^* = \{v_{16}\}$ (Line 13-15), i.e., $(node(v_5), node(v_{16}))$ and $(node(v_{15}), node(v_{16}))$, as marked by blue dashed lines.*

Next we focus on the update operations based on C (Line 16-29). In order to update T_1 through only the necessary operations, we scan the nodes of C layer-by-layer in the tree in a bottom-up manner, i.e., from nodes with large coreness to the ones with small coreness (Line 16). Here k_{max} represents the largest coreness of a vertex in G^*. For the coreness value K in one iteration, we select an unvisited node n_0 in C with $core(n_0) = K$ in T (Line 17). We use $\mathbb{N}_1$ to store each visited node $n_1 \in C$ with coreness K (Line 18 and 23), where the nodes will be merged into one node, W.l.o.g, n_0, at Line 28. Besides, $\mathbb{N}_2$ records the parent node of each node in $\mathbb{N}_1$, and each node n_2 if there is a node pair $(n_1, n_2) \in C$ and $core(n_2) < K$ (Line 20 and 25). Each node in $\mathbb{N}_2$ will become (part of) an ancestor of n_0 on T^*. To maintain the tree structure, after collecting the nodes for $\mathbb{N}_1$ and $\mathbb{N}_2$, we pick a node $n' \in \mathbb{N}_2$ with the largest coreness. The node n' is actually a child node of other nodes in $\mathbb{N}_2$ with corenesses smaller than $core(n')$, and these nodes on the same layer will be merged into one node in later iterations. So, we add (n', n_2) to C for each $n_2 \in \mathbb{N}_2 \setminus n_2$. Then, we merge the nodes in $\mathbb{N}_1$ to one node (Line 28) and reset its parent node (Line 29). Iteratively, the k-core hierarchy is correctly maintained.

Example 5. *We use Figure 3(c)(d) to illustrate the mergence procedure (Line 16-29 of Algorithm 2) on the case in Figure 3(a). In the first iteration, $n_0 = node(v_1)$ is selected at Line 17 and pushed into $\mathbb{N}_1$ at Line 18. Then, $P(node(v_1)) = node(v_{15})$ is pushed into $\mathbb{N}_2$. After Line 25, we get $\mathbb{N}_1 = \{node(v_1)\}$ and $\mathbb{N}_2 = \{node(v_{15}), node(v_{14})\}$. We push $(node(v_{15}), node(v_{14}))$ into C at Line 27. In the second iteration of Line 16, if $n_0 = node(v_{15})$, we have $\mathbb{N}_1 = \{node(v_{15}), node(v_{14})\}$ and $\mathbb{N}_2 = \{node(v_{16}), node(v_5)\}$ after Line 25. If $n' = node(v_5)$ at Line 26, we push $(node(v_5), node(v_{16}))$ into C at Line 27. The nodes in $\mathbb{N}_1$ are merged at Line 28, and $P(node(v_{15})) = node(v_5)$ at Line 29. After all the iterations, we get T^* as shown in Figure 3(d).*

Theorem 6. *After running Algorithm 2, for each node $n_0 \in T_0$ if n_0 is not in C, we have (i) n_0 keeps the same in T^* and (ii) the child nodes of n_0 keeps the same in T^*.*

Proof. (i) Suppose $V(n_0)$ changed, there is a vertex in $V(n_0) \cap V^*$ moved out of n_0, or a vertex in V^* is moved to $V(n_0)$. In both cases, n_0 will be added to a node pair in C by Line 7, which contradicts with $n_0 \notin C$. (ii) Now we confirm that n_0 keeps the same as in T^*, i.e. $V^* \notin V(n_0)$. Thus no vertex is added to any child node of n_0. (ii.a)For a child of n_0 changed its parent node, there must be a vertex moved out of n_0, which contradicts with that n_0 keeps the same in T^*. (ii.b) Suppose there is a vertex moved out of a n_0's child node n_c, n_c will be added in C by Line 7, and thus (n_c, n_0) will be added to C by Line 26, which contradicts with $n_0 \notin C$. Thus, the theorem holds. □

Correctness. According to Theorem 6, the local structure of each node $n_0 \notin C$ keeps the same, i.e., its vertex set and child nodes are same in T_0 and T^*. Thus, we only need to process the nodes in C for the update of T_0. As we follow the definition of k-core hierarchy to maintain T_0, the correctness of Algorithm 2 is guaranteed.

Complexity. The space complexity of Algorithm 2 is $O(m)$. As Algorithm 2 combines with the operations in Algorithm 1 for each inserted edge, according to the time complexity of Algorithm 1, the worst-case time complexity of Algorithm 2 is $O(\sum |V^*| + x \cdot (k_{max} + |T_0|))$ where $|T_0|$ is the number of tree nodes in T_0.

4 EDGE REMOVAL

In this section, we first study the maintenance of k-core hierarchy against the removal of one edge. Then, we extend the study to the removal of multiple edges. Let (x_1, x_2) denote the edge to be removed from G_0, where $(x_1, x_2) \in E(G_0)$. W.l.o.g, we suppose $K = core(x_1) \leq core(x_2)$. Let G^* denote the graph updated from G_0, i.e., $G^* = (V, E - (x_1, x_2))$. Let $T_0 = T(G_0)$ and $T^* = T(G^*)$. When the context is clear, we use $node(v_0)$ to represent $node(v_0, T)$.

4.1 Removal Analysis

Similar to the case of edge insertion, we apply [66] to update the corenesses of all the affected vertices. For the update of k-core hierarchy against edge removal, we need to check the connectivity of the vertices in some tree nodes to examine whether a node will split or not. As the insertion of edges will not split the tree nodes, the cost of maintaining the k-core hierarchy against edge removal is generally higher than that against edge insertion. Thus, in order to reduce the cost, we should carefully limit the search space to address the connectivity change.

Let V^* denote the set of vertices with coreness changed after the removal of (x_1, x_2). Essentially, the removal of an edge (x_1, x_2) from G_0 is a reverse procedure of inserting (x_1, x_2) to G^*. So, we can immediately deduce the following rules.

- For every vertex $v \in V^*$, we have $core(v, G_0) = K$ and $core(v, G^*) = K - 1$.
- We have $V^* \subseteq V(C(x_1, G_0))$, and the induced subgraph of V^* in G_0 is connected.

Similar to the analysis for insertion, for the update of k-core hierarchy with removing (x_1, x_2), we discuss the following cases for all the k-cores in G_0.

(i) The k-cores with $k > K$. For every vertex v with $core(v) > K$, we have $C(v, G^*) = C(v, G_0)$, because $(x_1, x_2) \notin C(v, G_0)$. Thus, the hierarchy of k-cores (the subtrees rooted on L_k) with $k > K$ keeps the same in G_0 and G^*.

(ii) The k-cores with $k \leq K$. For every vertex v with $core(v) < K$, we have $core(v, G^*) = core(v, G_0)$. The removal of (x_1, x_2) will move the vertices in V^* to L_{K-1} from L_K. Besides, the ancestors of $node(x_1)$ or $node(x_2)$ may split, because some k-cores become disconnected by the removal of (x_1, x_2) and the move of the vertices in V^*.

Let n' denote the node in T_0 containing V^*. Note that the vertices in V^* are actually in one node of T_0 before the removal of (x_1, x_2); otherwise Theorem 3 is violated if we insert (x_1, x_2) back. As $core(x_1, G_0) \leq core(x_2, G_0)$ is supposed, we have $V^* \subseteq V(node(x_1, T_0))$.

4.2 Adjust the Subtree rooted at n'

As shown in above subsection, the hierarchy of k-cores keeps the same when $k > K$. Here, we show how to adjust the subtree rooted at n' in T_0. We will split the ancestors of n' in next subsection.

Algorithm 3 shows the pseudo-code to maintain the k-core hierarchy against the removal of (x_1, x_2). We first compute the set V^* at Line 2, where the coreness of each vertex decreases to $K - 1$ from K, and get n' which contains V^* at Line 3. If the parent node of n' is not in L_{K-1}, we create a new child node of $P(n')$ in L_{K-1} and set the parent node of n' to it (Line 7-9). Let n^* denote the parent node of n' (Line 4 or 8), we move V^* to n^* from n' in Line 10.

Split n'. As the move of V^* may disconnect $G^*[n']$, we need to find all connected components of it in Line 12. For each child node n_c of n', we make sure that $G^*[n_c]$ is already a complete $(K+1)$-core otherwise Theorem 4 is violated if we insert (x_1, x_2) back. Thus, instead of traversing all vertices in $G^*[n']$, we can regard the vertices in $T'(n_c)$ as a unit for each child node n_c of n'. We will describe it in details later. In this way, we can immediately find the vertices in $V(n')$ which should exist in the same node of $L_K(T^*)$, and its child node on T^*.

```
Algorithm 3: RemoveOne
Input  : a graph $G_0$, the $k$-core hierarchy $T_0$, an edge
         $(x_1, x_2) \in E(G_0)$
Output : $T^*$, i.e., the updated $T_0$
1  $T \leftarrow T_0$;
2  $V^* \leftarrow$ vertices with coreness changed by removing $(x_1, x_2)$ from $G_0$;
3  $n' \leftarrow$ node$(x_1)$ in $T$ (suppose $core(x_1) \leq core(x_2)$);
4  $n^* \leftarrow P(n')$;
5  if $V^* \neq \emptyset$ then
6      if $core(P(n')) \neq K - 1$ then
7          create $n_0$ on $L_{K-1}$ as a child node of $n^*$ ;
8          $n^* \leftarrow n_0$;
9          $P(n') \leftarrow n^*$;
10     move each vertex in $V^*$ from $n'$ to $n^*$;
11 $T_1 \leftarrow T$;
12 $T_2 \leftarrow$ SplitNode$(n', T_1)$;
13 flag $\leftarrow$ true;
14 $i \leftarrow 2$;
15 while flag = true do
16     $i \leftarrow i + 1$; $n_i^* = P(n_{i-1}^*)$;
17     $T_i \leftarrow$ SplitNode$(n_i^*, T_{i-1})$;
18     flag $\leftarrow (T_{i-1} \neq T_i)$;
19 $T^* \leftarrow T_i$;
20 return $T^*$
```

```
Algorithm 4: SplitNode
Input  : a subtree rooted at $n_r$ to split, the $k$-core hierarchy $T$
Output : the updated $T$
$n_r^* \leftarrow P(n_r)$; $V_r \leftarrow V(n_r)$; $K = core(n_r)$;
for each vertex $u \in V(n_r)$ do
    create an empty node $n_c$ on $L_K$ as a child node of $n_r$;
    move $u$ to $n_c$ from $n_r$;
for each node $n_c \in n_r$.children do
    for each node $n_d \in T'(n_c)$ do
        $cn(n_r, n_d) \leftarrow n_c$;
for each vertex $u \in V_r$ do
    for each vertex $v \in N(u, G^*)$ do
        if $core(v, G^*) = K$ then
            merge $node(u)$ and $node(v)$;
        else if $core(v, G^*) > K$ then
            $n_c \leftarrow cn(n_r, v)$ ;              /* FindSubroot($n', v$) */
            if $P(n_c) = n_r$ then
                $P(n_c) \leftarrow node(u)$;
            else
                merge $node(u)$ and $P(n_c)$;
for each node $n_c \in n_r$.children do
    $P(n_c) \leftarrow n_r^*$;
remove $n_r$ from $T$;
return $T$, i.e., updated $T$
```

Algorithm 5: **FindSubroot**

```
  Input  : a node $n_0$, a vertex $v_0$
  Output : the node $n_c$, i.e., $cn(n_0, v_0)$
1 $A \leftarrow$ empty set; $n_c \leftarrow n_1 \leftarrow node(v_0)$;
2 while $n_1 \neq n_0$ do
3     $A \leftarrow A \cup \{n_1\}$;
4     $n_c \leftarrow n_1$; $n_1 \leftarrow Jump(n_1)$;
5 $Jump(n_2) \leftarrow n_c$ for each node $n_2 \in \{A \setminus n_c\}$;
6 return $n_c$
```

Algorithm 4 shows the process of splitting the subtree rooted at n_r. Let n_r^* denote the parent node of n_r, and V_r denote the vertex set of n_r (Line 1). We first logically split n_r by regarding each vertex as a single child node of n_r^* in $L_{core(n_r)}$ (Line 2-4). The node n_r will become an empty node while the existing parent-child relations of n_r are temporally preserved. In the implementation, we mark the nodes and address the split together later to reduce cost.

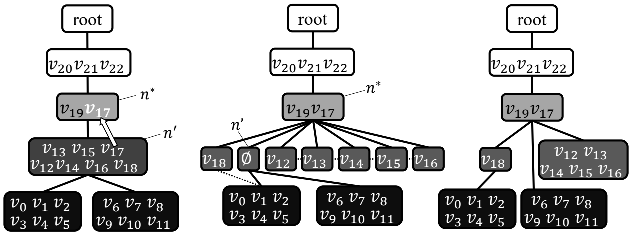
Use $cn(n_0, v_0)$. In order to fast check whether two vertices are in a same K-core on G^*, we use $cn(n_0, v_0)$ which has been discussed above. Note that $cn(n_0, v_0) = cn(n_0, node(v_0))$. For an edge $(u, v) \in E(G^*)$ with $u \in V_r$, (i) if $core(v) = K$, we merge $node(u)$ and $node(v)$ as they are linked by (u, v) (Line 10-11); (ii) if $core(v) > K$, we can retrieve that $n_c = cn(n_0, v_0)$ is a child node of $node(u)$, and thus there is a node containing $P(n_c)$ and $node(u)$ on L_K (Line 12-17). When $P(n_c) = n_r$, we just set $P(n_c)$ as $node(v)$ because n_r is an empty set now with some parent-child relations temporarily preserved (Line 14-15); otherwise, we merge $P(n_c)$ and $node(u)$ as their vertices are in the same node on L_K (Line 16-17).

After running Line 8-17 of Algorithm 4, for each child node n_c of n' left, n_c is not a child node of any node on $L_K \setminus n'$, and we set $P(n_c) = n^*$ (Line 18-19). Finally, we remove n' and its associated tree edges from T' at Line 20, and the split process is completed.

Example 6. *Consider the graph in Figure 1(a). Suppose an edge (v_2, v_{17}) is removed from the graph. In the process of maintaining k-core hierarchy, after we move $V^* = \{v_{17}\}$ to n^* and get T_1 in figure 4(a), figure 4(b) shows the process of spliting n'(Line 12 of Algorithm 3). For each vertex in $V(n')$, we first logically create a node at L_K as a child of $n^* = node(v_{19})$) and move the vertex to the new node (Line 1-4 of Algorithm 4), as shown in the left part of Figure 4(b). Now n' becomes an empty node with tree edges temporarily preserved.*

At Line 5-7 of Algorithm 4, we initialize $cn(n', n_d)$ to fast retrieve the child node n_c of n' which is an ancestor of n_d (or itself), e.g., $cn(node', v_0) = node(v_0)$. Then, we can traverse each vertex at L_K and visit its neighbors s.t. the nodes to merge and the tree edges to adjust can be immediately determined. For instance, we set $P(node(v_0)) = node(v_{18})$ at Line 15, because v_0 is a neighbor of v_{18} and $cn(node', v_0) = node(v_0)$. After the traversal, as $node(v_6)$ is not visited, we set $P(node(v_6)) = node(v_{17})$ at Line 18-19. The updated T_0 is shown on the right part of Figure 4(b).

Implementation of Computing $cn(n_0, v_0)$. The pseudo-code to compute $cn(n_0, v_0)$ is shown in Algorithm 5. It is not necessary to generate $cn(n_0, n_d)$ for every descendent n_d of n_c. In the implementation, we use a global pointer $Jump(node(v_0))$ to compute and preserve $cn(n_0, v_0)$ in a lazy manner, i.e., only when it is required at Line 13 of Algorithm 4. The pointer $Jump(n_0)$ is initialized by $P(n_0)$ for each node n_0 to facilitate the search. We use A to record the visited nodes in the search, i.e., some ancestors of $node(v_0)$. The currently visited node is denoted by n_1, and the last visited node is denoted by n_c. They are initialized at Line 1. We search the subtree rooted at n_0 layer-by-layer in a bottom-up manner, starting from $n_1 = node(v_0)$ until $n_1 = n_0$ (Line 2). Each visited node is pushed into A at Line 3. Then, n_c is set by n_1, and n_1 is set by $Jump(n_1)$ at Line 4. When $n_1 = n_0$, the node n_c is the child node of n_0 with $T'(n_c)$ containing v, i.e., $cn(n_0, v_0) = n_c$. For each node n_0 in A, we have $cn(n_0, n_0) = n_c$, and thus $Jump(n_0)$ is set by n_c at Line 5 (except $n_0 = n_c$ to maintain stop condition). Finally, we return n_c.



(a) T_1

(b) SplitNode(n')

Figure 4: Split n' in T_1

4.3 Split Ancestors of n'

After adjusting the subtree rooted at n' (Line 1-10 of Algorithm 3), the parent node of n', i.e., n^* may also split. The split process may further spread to the ancestors of n'.

We will continue to try to split the node n^* as V^* moves to n^*. The child nodes of n^* keep the same in T^* according to Theorem 2 and the correctness of Algorithm 4. The split procedure of n^* is essentially same to that of n', because the split spreads in layer-by-layer manner from bottom to up, we use *splitNode* to adjust the subtree rooted at n^* (Line 15-18). After processing n^*, we iteratively set its parent node as the next node to split (Line 16-17). Once the tree does not change (Line 18), the split stops.

For any other node which has not been an input of SplitNode (Algorithm 4), the following theorem holds.

Theorem 7. *After running Algorithm 3, for any node n_0 in T_0 which has not been an input of SplitNode (Algorithm 4), n_0 keeps the same in T^* and the child nodes of n_0 keep the same in T^*.*

Proof. For the removal of (x_1, x_2), as only the vertices in n' move to n^*, the vertex set of each layer L_k keeps the same except for $k = K$ or $K-1$. (i) Suppose $V(n_0)$ changes. We have $n_0 \notin \{n', n^*\}$ as n' (resp. n^*) is an input of *SplitNode* in Line 9 (resp. Line 12). As $V(n_0)$ changes and $n_0 \notin \{n', n^*\}$, $G^*[n_0]$ must be disconnected in G^*. For each k-core $C_0^k \subseteq G_0[n_0]$, $(x_1, x_2) \in C_0^k$, and $k \leq K$, we have C_0^k is also disconnected in G^*; otherwise, $G_0[n_0]$ will not split. Then, in Line 11-14 of Algorithm 3, n_0 will be an input of *SplitNode* which causes a contradiction. Thus, n_0 keeps the same in T^*.

(ii) Suppose the child nodes of n_0 are different in T and T^*. If $V^* = \emptyset$, the child nodes of n_0 split, and the split stops at n_0 as n_0 keeps the same. Then n_0 is an input of *SplitNode* according to Line 13. If $V^* \neq \emptyset$, when n^* is a child node of n_0, n_0 is also an input of *SplitNode* according to Line 13; when n' is a child node of n_0, we have $n_0 = n^*$ or $n_0 = P(n^*)$ in T^* (Line 6), n_0 is an input; for other

Algorithm 6: **RemoveX**

```
    Input  : a graph $G_0$, the $k$-core hierarchy $T_0$, an edge set $E' \subseteq E(G_0)$
    Output : $T^*$, i.e., the updated $T_0$
1   $T \leftarrow T_0$; $G \leftarrow G_0$; $C \leftarrow \emptyset$;
2   for each $(u, v) \in E'$ do
3       $V^* \leftarrow$ vertices with coreness changed by removing $(u, v)$ from $G$;
4       $G \leftarrow G - (u, v)$;
5       $node' \leftarrow node(u, T_i)$ (suppose $K = core(u, G) \leq core(v, G)$);
6       if $core(P(node')) = K - 1$ then
7           $node^* \leftarrow P(node')$;
8       else
9           create an empty node $node^*$ on $L_{K-1}$ as a child of $P(node')$;
10          $P(node') \leftarrow node^*$;
11      move each vertex $v \in V^*$ from $node'$ to $node^*$;
12      $C \leftarrow C \cup \{node', node^*\}$;
13  $T_1 \leftarrow T$; $G^* \leftarrow G$; $i = 1$;
14  for each $n' \in C$ in descending order of coreness do
15      $i \leftarrow i + 1$;
16      $T_i \leftarrow$ SplitNode$(n', T_{i-1})$;
17      $C \leftarrow C \cup \{P(n')\}$ if $T_{i-1} \neq T_i$;
18  return $T$, i.e., $T^*$
```

cases, the split stops at n_0 as n_0 keeps the same in T^*. Thus, n_0 is an input of *SplitNode*, which causes a contradiction. □

Correctness. According to Theorem 2, Theorem 3 and the definition of the k-core hierarchy, the correctness from T_0 to T_1 is guaranteed. By Theorem 7, the local structure of each node n_0, which has not been an input of *SplitNode*, keeps the same, i.e., its vertex set and child nodes are same in T_1 and T^*. So, we iteratively execute *SplitNode* for the update of T_1. As we follow the definition of the hierarchy to maintain T_1, the correctness of Algorithm 3 is guaranteed.
Complexity. We show it in Section 4.4, as Algorithm 3 is essentially same to the update algorithm for removing x edge(s) when $x = 1$.

4.4 Remove x Edges

In this section, we update the k-core hierarchy once for the removal of x edges. Let E' denote the edge set to be removed from G_0, where $E' \subseteq E(G_0)$. Algorithm 6 shows the pseudo-code to maintain the k-core hierarchy against the removal of E'. Similar to Algorithm 2 and in the way of Algorithm 3, for each edge $(u, v) \in E'$, we remove it from G (the changing graph), compute V^* at Line 3, and adjust the parent node of $node'$ (Line 5-10) where $node' = node(u)$ with $core(u, G) \leq core(v, G)$. If $P(node')$ is at $L_{core(u)-1}$, we set $node^*$ by $P(node')$ (Line 6-7); otherwise, we create a node $node^*$ at $L_{core(u)-1}$ as a child node of $P(node')$, and reset $P(node')$ by $node^*$ (Line 8-10). We move the vertices in V^* to $node^*$ (Line 11).

We use a candidate set C to store all the nodes that should be visited by *SplitNode*. Due to the removal of some edges and/or the coreness change of some vertices, (i) a node n' may split and/or (ii) a child node of n' may change its parent node to $P(n')$, i.e., the subtree rooted at n' may change. Thus, we push every such node n' into C at Line 12 and 17. It is correct to split a node n' in C from large coreness to small (by Algorithm 4), as the subtree of each child node of n' is already updated (Line 14). We make sure that, after changing k-core hierarchy to T_i, each node in current C is not changed between T_{step2} and T_i. After processing all the nodes in C, there is no other node to split, and we get T^*.

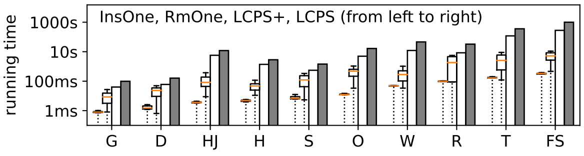
Table 2: Statistics of Datasets

Dataset	$\lvert V\rvert$	$\lvert E\rvert$	d_{avg}	k_{max}	$\lvert T\rvert$
Gowalla	196,591	950,327	9.7	51	75
DBLP	317,080	1,049,866	6.6	113	767
Human-Jung	784,262	267,844,669	683.1	1200	4088
Hollywood	1,069,126	56,306,653	105.3	2208	679
Skitter	1,696,415	11,095,298	13.1	131	903
Orkut	3,072,441	117,185,083	76.3	253	254
Wiki	12,150,976	378,142,420	62.2	1122	5049
Rgg	16,777,216	132,557,200	15.8	20	117422
Twitter	41,652,230	1,468,365,182	8.8	2488	3049
Friend**S**ter	65,608,366	1,806,067,135	55.1	304	451

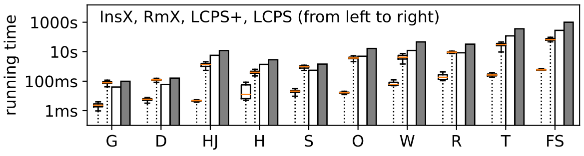
Correctness. Splitting the nodes in C from bottom to up is essentially same to the split process of removing one edge (Algorithm 3). Thus, by Theorem 7, the correctness of Algorithm 6 is guaranteed.
Complexity. The space complexity of Algorithm 6 is $O(|E|)$. The time cost of Algorithm 6 is the sum of two parts. The first part is to maintain each vertex's coreness and to move the vertices in V^* to the correct layers, which costs $O(\sum_{w \in V^*} |N(w)| + \sum_{w \in V^*} |N(w)| \times log|O_K| + |V^*| \times log|O_{K-1}| + |V^*| + |\cup_{v \in V^*} node(v)|)$ for each edge need to be deleted. The second part is to split all nodes in C in Line 14-17. For each node n' as the input of Algorithm 4, we visit the neighbors of each vertex and maintain $cn(\cdot)$ for each visited descendant of it, which takes $O(\sum_{v \in V(n')} |N(v, G^*)| + |T'(n')|)$. Time complexity of Algorithm 6 is the sum of the above two parts.

5 EXPERIMENTAL EVALUATION

Datasets. In the experiments, we use 10 public real-world networks with size up to billion-scale. The datasets are from different areas including collaboration networks, Internet topology, brain networks, and social networks. Hollywood, Human-Jung, and Rgg[47] can be downloaded from http://networkrepository.com; Twitter [28] is from http://an.kaist.ac.kr/traces/WWW2010.html; and the rest are from http://snap.stanford.edu/data. Table 2 shows the statistics of the datasets where d_{avg} is the average vertex degree, k_{max} is the largest coreness and $|T|$ is the number of nodes in k-core hierarchy.
Algorithms. We evaluate the update algorithms against the insertion/removal of one edge, i.e., Algorithm 1 and 3, denoted by *InsOne* and *RmOne*, respectively. We also evaluate the update algorithms to address a batch of inserted/removed edges, i.e., Algorithm 2 and 6, denoted by *InsX* and *RmX*, respectively. The state-of-the-art algorithm for building the k-core hierarchy T is LCPS [42, 51] (denoted by *LCPS*). It can be used to compute the hierarchy from scratch on dynamic graphs. Given a set E' of edges to be inserted/removed, an improved baseline is denoted by *LCPS+*, where we use the state-of-the-art algorithm to update the coreness of each vertex [66], and apply LCPS to compute the k-core hierarchy of the connected component(s) containing the endpoints of E'.
Environment. We perform the experiments on a CentOS Linux server with Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GH, and 256G memory. All the algorithms are implemented in C++. The source code is compiled by GCC under O3 optimization.

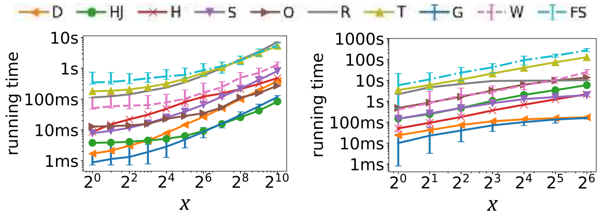


(a) single edge



(b) 10 edges

Figure 5: Performance on All the Datasets



(a) Insertion (InsX) (b) Removal (RmX)

Figure 6: Performance on Inserting/Removing x Edges

5.1 Performance on Runtime

In the evaluation of performance, for each test case, we randomly remove x edge(s) in each graph for RmOne ($x = 1$) and RmX, and insert the edges back for InsOne and InsX. We execute the algorithms by 100 independent cases and report the running time of one case in average.

Inserting/Removing One Edge. We first evaluate InsOne for the insertion of one edge, and RmOne for the removal of one edge, compared with LCPS+ as introduced at the beginning of Section 5. Figure 5(a) shows the running time of four algorithms on all the datasets. As our algorithms well capture a small part of T that will be changed, the runtime is more fluctuant than LCPS and LCPS+. Thus, we show the standard box-plot of InsOne (RmOne) on each dataset where the red line represent the median runtime. Our incremental algorithms significantly outperform LCPS/LCPS+ on different scales of graphs, by up to 3100 times for InsOne and up to 270 times for RmOne. The running time of RmOne is generally larger than that of InsOne, because some tree nodes may split due to the removal and we have to check whether a k-core is still connected. In real-life data, the case of insertion is more important, as it is usually more frequent than removal. Besides, the outperformance of our algorithms can be better on larger datasets, because the insertion/removal of x edges affects T less on larger graphs. InsOne and RmOne can also be iteratively executed to update T instantly. It is quite promising to apply InsOne and RmOne against graph dynamics.

Inserting/Removing x Edges. If a low update frequency is acceptable, we can update T once until x edges are received to be inserted/removed. InsX is used for the insertion of x edges and RmX is used for the removal. The results are reported in Figure 5(b) when $x = 10$. InsX and RmX are faster than the baselines on most datasets, especially for large data. Note that the k-core hierarchy may change a lot for such x values on small datasets. For instance, when $x = 50$, the nodes in C (the candidates to split) of RmX occupy more than 95% of $|V(G)|$ on small graphs (e.g., G and D).

Table 3: The engagement of users in node n_1, compared with the parent node of n_1 (T-edge), or the nodes with smaller subtrees (T-size), on DBLP from Year 19-20 (Win Percent)

k	1	2	3	4	5	6	7	8	total
T-edge(%)	-	100	99.7	98.9	100	100	100	100	99.44
T-size(%)	80.9	78.6	86.2	93.4	80.6	44.1	100	100	84.58

Figure 6 reports the trends of InsX and RmX with different x values. As edge insertion is more frequent than edge removal in real-life, the x values are from 1 to 1024 for InsX, and from 1 to 64 for RmX, respectively. For very large x values, we recommend to rebuild the hierarchy, or apply InsOne and RmOne for instant update. When x is small, the affected area of T is very small for both insertion and removal. For insertion, as only a few number of nodes are affected, the fluctuation is small. However, for removal, we need to traverse the vertices in the whole affected node for connectivity check which causes a larger fluctuation. When x becomes larger, the affected area of T is larger. For insertion, the number of node pairs in C may be large (whenever they satisfy the relationship in T_1 or not). In some cases, many node pairs in C are already in correct parent-child relations (no adjustment is required), which causes a large fluctuation. However, for removal, the fluctuation is small, because almost all the nodes in the candidate set will be traversed for connectivity check. When x is even larger in Figure 6, the fluctuation becomes smaller because a large portion of the graph and the hierarchy are visited.

Runtime Analysis on Different Cases. For different orders of insertion/deletion on a same set of edges, the runtime difference of maintenance is too small to observe. In Figure 5, the main factors to affect the runtime are $|V(G)|$, $|E(G)|$ and $|E'|$. For insertion, the effect of $|V(G)|$ may relate to the size of V^*, as the size of V^* is relatively proportional to $|V(G)|$. For deletion, the effect of $|V(G)|$ is less obvious as shown in Figure 5(b). Figure 5 shows that $|E(G)|$ also largely affects the performance, e.g., HJ and H. We also test our algorithms by removing top-100 edges with the highest/lowest betweenness centrality scores and then inserting them back, while the overall cost of runtime is not affected by different scores.

5.2 Application on User Engagement Analysis

The status of user engagement is a key indicator of a network. The existing works use the coreness of a vertex to estimate its engagement level [35]. Here we investigate the engagement of the authors (users) and their characteristics in the k-core hierarchy of the coauthor graph from DBLP data in 2019 and 2020 [15]. Each author is a vertex and two authors are connected if they coauthored in a paper as the first 5 authors (to avoid noise from a paper with many authors). The engagement of a tree node is the average number of papers published by the users in the node. We compute the percentage of the users in the k^{th} layer (L_k) which is in a node with higher activity than its parent node, denoted by T-edge for each k value. Let n_m denote the node in L_k with the largest subtree, i.e., $T'(n_m)$. We compute the percentage of the users in L_k which is in a subtree (k-core) smaller than $T'(n_m)$ and with smaller activity than n_m, denoted by T-size for each k value. Table 3 shows that both T-edge and T-size are close to 100% for most k values. It implies

Table 4: Finding the densest subgraph by extracting a k-core

dataset	CoreApp		Opt-D (output S^*)			
				Opt-D+	Ins-D	Rm-D
	density	time(s)	density	time(s)		
G	76	0.2	87.59	0.06	<0.01	<0.03
D	113	0.25	113.13	0.07	<0.01	<0.05
HJ	2013.88	15.27	2114.92	6.54	<0.01	<0.58
H	2208	3.64	2208	1.60	<0.06	0.01-0.12
S	150.02	1.6	178.8	0.69	<0.02	<0.34
O	438.64	24.17	455.73	9.56	0.01-0.02	0.03-1.06
W	1142.43	61.9	1200.88	17.25	0.04-0.11	0.03-3.29
R	21.41	47.81	27.43	10.29	0.09-0.29	0.08-5.60
T	2873.15	448.6	3286.51	134.44	0.16-0.24	0.12-8.95
FS	513.85	1249.83	547.04	343.02	1.88-2.27	0.46-20.86

that the engagement evaluation of a vertex can be more accurate by considering both its coreness and its position in k-core hierarchy.

5.3 Application on Cohesive Subgraph Mining

Densest Subgraph. Finding the densest subgraph (DS) on static graphs is a fundamental NP-hard problem in graph analytics [20], which aims to find the subgraph with the largest average vertex degree (i.e., density). Recently, a 0.5-approximate solution (Opt-D) is proposed in [11] by extracting a k-core in T with the largest density, whose output is denoted as S^*. The previous state-of-the-art approximate solution is CoreApp proposed in [20].

A baseline solution is Opt-D+ which first updates T by LCPS+, and then uses Opt-D to compute DS . In this experiment, we apply our algorithms to Opt-D, denoted by Ins-D and Rm-D, to maintain DS against edge insertion and removal, respectively. In Ins-D and Rm-D, we first maintain T by our algorithms (i.e., InsOne and RmOne), and mark each node whose vertex set changed or child node set changed during the update of T. Then, we run Opt-D on the subtrees of T containing the marked nodes, to update DS.

Table 4 shows that the solution S^* produced by the algorithms based on Opt-D has a higher density than that from CoreApp. The outperformance is similar on dynamic graphs. The runtime of Opt-D+ on dynamic graphs is much faster than the re-computation from scratch (i.e., CoreApp). As our Ins-D and Rm-D efficiently update the k-core hierarchy, the runtime is smaller than Opt-D+ by up to 3 orders of magnitude.

Maximum Clique. Given a graph G, the maximum clique (MC) problem is to find the largest subgraph of G such that every pair of vertices in the subgraph are adjacent [5]. Let $MC(S)$ denote the size of the maximum clique on subgraph S. As shown in Table 5, the maximum clique on S^* (the result from Ins-D and Rm-D) well approximates the maximum clique on G on most datasets, although the size of S^* is less than 1.2% of G on all the datasets. This finding benefits the algorithm design for MC problem on dynamic graphs.

6 RELATED WORK

We review more works besides those in the introduction. Many cohesive subgraph models are proposed to accommodate different scenarios, e.g., clique [10], quasi-clique [45], nucleus [51], k-core [3, 11, 26, 36, 52], k-truss [13, 25, 38, 55, 57], k-plex [59], and k-ecc [6, 68]. A graph can be decomposed into a hierarchical structure by some of the models, e.g., core decomposition [35, 37, 43, 60], truss decomposition [53, 55, 67], and ecc decomposition [6, 62].

Table 5: Finding the maximum clique by shrinking a k-core

datasets	G	D	HJ	H	S	O	W	R	T	FS
$\frac{MC(S^*)}{MC(G)}(\%)$	60	100	100	100	87	57	100	100	98.1	17
$\frac{\lvert S^*\rvert}{\lvert G\rvert}(\%)$	0.28	0.04	1.15	0.21	0.03	0.85	<0.01	<0.01	0.01	0.07

Core decomposition is one of the most well-studied models, due to its effectiveness in various applications including community discovery [7, 8, 22, 29, 30, 32, 56, 58, 63], influential spreader identification [17, 27, 34, 40], and network analysis [1, 14, 23, 54]. Core decomposition is surveyed in [39].

The model of k-core is often used to find high-quality communities, where the connectivity is often required for modeling a community, e.g., (k, r)-core [65], diversified coherent k-core [70], persistent k-core [32], temporal k-core [61], and skyline k-core [30]. The k-core hierarchy T can be used as an effective index to speed up the community discovery, e.g., [31]. Recently, a time and space optimal solution is proposed to find the best k-core subgraphs in the k-core hierarchy [11].

An in-memory algorithm for core decomposition is proposed in [3], with a time complexity of $O(m)$. The k-core hierarchy can also be constructed in $O(m)$ time [42]. Core decomposition has been studied under different configurations, including distributed environment [43], graph stream [50], parallel setting [21], and Map-Reduce [46]. For graphs that are too large to fit in the memory, an I/O efficient algorithm for core decomposition is proposed in [60], and EM-Core [9] is an external algorithm that runs in a top-down manner. Core decomposition of large graphs on a single PC is studied using `GraphChi`, `WebGraph`, and external model [26]. The pressing problems in large graph processing are surveyed in [48].

7 CONCLUSION AND FUTURE WORK

Due to the wide applications of core decomposition and the fast evolving of real-world graphs, in this paper, we study the problem of maintaining the k-core hierarchy on dynamic graphs. Through rigorous theoretical analyses, we propose effective local update techniques. Our algorithms for updating the k-core hierarchy largely outperform the baselines for one or a small batch of updated edge(s). Our approach may be adapted to other decompositions if they hold the same hierarchical structure. Nevertheless, it may be non-trivial to design novel techniques if the connectivity issue becomes different to that in k-core. Besides, the framework of our algorithms may inspire a sound solution for parallel maintenance of k-core hierarchy. The first challenge is to construct the k-core hierarchy on static graphs in parallel. Then, when a set of edges are inserted, (i) the coreness of each vertex can be updated in parallel, e.g., [24], and (ii) for each k value in decreasing order from k_{max} to 0, each node in the k^{th} layer may be merged by one thread. The case of edge deletion is similar to the insertion, because the split of each node may be handled by one thread from the last layer to the root.

ACKNOWLEDGMENTS

This work is partially supported by the National Key R&D Program of China under grant 2018AAA0102502 and the National Natural Science Foundation of China under Grant U20B2046. Fan Zhang is also partially supported by NSFC62002073. Xuemin Lin is also partially supported by ARC DP180103096 and DP170101628. Wenjie Zhang is also partially supported by ARC DP180103096.

REFERENCES

[1] 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. *NHM* 3, 2 (2008), 371–393. https://doi.org/10.3934/nhm.2008.3.371

[2] Gary D. Bader and Christopher W. V. Hogue. 2003. An automated method for finding molecular complexes in large protein interaction networks. *BMC Bioinformatics* 4 (2003), 2. https://doi.org/10.1186/1471-2105-4-2

[3] Vladimir Batagelj and Matjaz Zaversnik. 2003. An O(m) Algorithm for Cores Decomposition of Networks. *CoRR* cs.DS/0310049 (2003).

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

[5] Lijun Chang. 2019. Efficient Maximum Clique Computation over Large Sparse Graphs. In *KDD*. 529–538. https://doi.org/10.1145/3292500.3330986

[6] Lijun Chang, Jeffrey Xu Yu, Lu Qin, Xuemin Lin, Chengfei Liu, and Weifa Liang. 2013. Efficiently computing k-edge connected components via graph decomposition. In *SIGMOD*. 205–216. https://doi.org/10.1145/2463676.2465323

[7] Lu Chen, Chengfei Liu, Kewen Liao, Jianxin Li, and Rui Zhou. 2019. Contextual Community Search Over Large Social Networks. In *ICDE*. 88–99. https://doi.org/10.1109/ICDE.2019.00017

[8] Lu Chen, Chengfei Liu, Rui Zhou, Jianxin Li, Xiaochun Yang, and Bin Wang. 2018. Maximum Co-located Community Search in Large Scale Social Networks. *PVLDB* 11, 10 (2018), 1233–1246. https://doi.org/10.14778/3231751.3231755

[9] James Cheng, Yiping Ke, Shumo Chu, and M. Tamer Özsu. 2011. Efficient core decomposition in massive networks. In *ICDE*, Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan (Eds.). IEEE Computer Society, 51–62. https://doi.org/10.1109/ICDE.2011.5767911

[10] James Cheng, Yiping Ke, Ada Wai-Chee Fu, Jeffrey Xu Yu, and Linhong Zhu. 2011. Finding maximal cliques in massive networks. *ACM Trans. Database Syst.* 36, 4 (2011), 21:1–21:34. https://doi.org/10.1145/2043652.2043654

[11] Deming Chu, Fan Zhang, Xuemin Lin, Wenjie Zhang, Ying Zhang, Yinglong Xia, and Chenyi Zhang. 2020. Finding the Best k in Core Decomposition: A Time and Space Optimal Solution. In *ICDE*. 685–696. https://doi.org/10.1109/ICDE48307.2020.00065

[12] Aaron Clauset, Cristopher Moore, and Mark EJ Newman. 2008. Hierarchical structure and the prediction of missing links in networks. *Nature* 453, 7191 (2008), 98.

[13] Jonathan Cohen. 2008. Trusses: Cohesive subgraphs for social network analysis. *National Security Agency Technical Report* 16 (2008), 3–1.

[14] Madelaine Daianu, Neda Jahanshad, Talia M. Nir, Arthur W. Toga, Clifford R. Jack Jr., Michael W. Weiner, and Paul M. Thompson. 2013. Breakdown of Brain Connectivity Between Normal Aging and Alzheimer's Disease: A Structural *k*-Core Network Analysis. *Brain Connectivity* 3, 4 (2013), 407–422. https://doi.org/10.1089/brain.2012.0137

[15] The dblp team. 2020. Monthly snapshot release. In *DBLP computer science bibliography*. https://dblp.org/xml/release

[16] Yon Dourisboure, Filippo Geraci, and Marco Pellegrini. 2009. Extraction and classification of dense implicit communities in the Web graph. *TWEB* 3, 2 (2009), 7:1–7:36. https://doi.org/10.1145/1513876.1513879

[17] Sarah Elsharkawy, Ghada Hassan, Tarek Nabhan, and Mohamed Roushdy. 2017. Effectiveness of the k-core nodes as seeds for influence maximisation in dynamic cascades. *International Journal of Computers* 2 (2017).

[18] Wenfei Fan, Jianzhong Li, Jizhou Luo, Zijing Tan, Xin Wang, and Yinghui Wu. 2011. Incremental graph pattern matching. In *SIGMOD*. 925–936. https://doi.org/10.1145/1989323.1989420

[19] Yixiang Fang, Reynold Cheng, Xiaodong Li, Siqiang Luo, and Jiafeng Hu. 2017. Effective community search over large spatial graphs. *PVLDB* 10, 6 (2017), 709–720. https://doi.org/10.14778/3055330.3055337

[20] Yixiang Fang, Kaiqiang Yu, Reynold Cheng, Laks V. S. Lakshmanan, and Xuemin Lin. 2019. Efficient Algorithms for Densest Subgraph Discovery. *PVLDB* 12, 11 (2019), 1719–1732. https://doi.org/10.14778/3342263.3342645

[21] Mohsen Ghaffari, Silvio Lattanzi, and Slobodan Mitrovic. 2019. Improved Parallel Algorithms for Density-Based Network Clustering. In *ICML*. 2201–2210. http://proceedings.mlr.press/v97/ghaffari19a.html

[22] Christos Giatsidis, Fragkiskos D. Malliaros, Dimitrios M. Thilikos, and Michalis Vazirgiannis. 2014. CoreCluster: A Degeneracy Based Graph Clustering Framework. In *AAAI*. 44–50. http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8530

[23] Laurent Hébert-Dufresne, Antoine Allard, Jean-Gabriel Young, and Louis J Dubé. 2013. Percolation on random networks with arbitrary k-core structure. *Physical Review E* 88, 6 (2013), 062820.

[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 Distributed Syst.* 31, 6 (2020), 1287–1300. https://doi.org/10.1109/TPDS.2019.2960226

[25] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying k-truss community in large and dynamic graphs. In *SIGMOD*. 1311–1322. https://doi.org/10.1145/2588555.2610495

[26] Wissam Khaouid, Marina Barsky, S. Venkatesh, and Alex Thomo. 2015. K-Core Decomposition of Large Networks on a Single PC. *PVLDB* 9, 1 (2015), 13–23. https://doi.org/10.14778/2850469.2850471

[27] Maksim Kitsak, Lazaros K Gallos, Shlomo Havlin, Fredrik Liljeros, Lev Muchnik, H Eugene Stanley, and Hernán A Makse. 2010. Identification of influential spreaders in complex networks. *Nature physics* 6, 11 (2010), 888.

[28] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *WWW* (Raleigh, North Carolina, USA). ACM, New York, NY, USA, 591–600. https://doi.org/10.1145/1772690.1772751

[29] Conggai Li, Fan Zhang, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2019. Efficient Progressive Minimum k-core Search. *PVLDB* 13, 3 (2019), 362–375. https://doi.org/10.14778/3368289.3368300

[30] Rong-Hua Li, Lu Qin, Fanghua Ye, Jeffrey Xu Yu, Xiaokui Xiao, Nong Xiao, and Zibin Zheng. 2018. Skyline Community Search in Multi-valued Networks. In *SIGMOD*. 457–472. https://doi.org/10.1145/3183713.3183736

[31] Rong-Hua Li, Lu Qin, Jeffrey Xu Yu, and Rui Mao. 2015. Influential Community Search in Large Networks. *PVLDB* 8, 5 (2015), 509–520. https://doi.org/10.14778/2735479.2735484

[32] Rong-Hua Li, Jiao Su, Lu Qin, Jeffrey Xu Yu, and Qiangqiang Dai. 2018. Persistent Community Search in Temporal Networks. In *ICDE*. 797–808. https://doi.org/10.1109/ICDE.2018.00077

[33] Rong-Hua Li, Jeffrey Xu Yu, and Rui Mao. 2014. Efficient Core Maintenance in Large Dynamic Graphs. *TKDE* 26, 10 (2014), 2453–2465. https://doi.org/10.1109/TKDE.2013.158

[34] Jian-Hong Lin, Qiang Guo, Wen-Zhao Dong, Li-Ying Tang, and Jian-Guo Liu. 2014. Identifying the node spreading influence with largest k-core values. *Physics Letters A* 378, 45 (2014), 3279–3284.

[35] Qingyuan Linghu, Fan Zhang, Xuemin Lin, Wenjie Zhang, and Ying Zhang. 2020. Global Reinforcement of Social Networks: The Anchored Coreness Problem. In *SIGMOD*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 2211–2226. https://doi.org/10.1145/3318464.3389744

[36] 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. https://doi.org/10.1145/3308558.3313522

[37] Boge Liu, Fan Zhang, Chen Zhang, Wenjie Zhang, and Xuemin Lin. 2019. CoreCube: Core Decomposition in Multilayer Graphs. In *WISE*. 694–710. https://doi.org/10.1007/978-3-030-34223-4_44

[38] Boge Liu, Fan Zhang, Wenjie Zhang, Xuemin Lin, and Ying Zhang. 2021. Efficient Community Search with Size Constraint. In *ICDE*.

[39] 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. https://doi.org/10.1007/s00778-019-00587-4

[40] Fragkiskos D Malliaros, Maria-Evgenia G Rossi, and Michalis Vazirgiannis. 2016. Locating influential nodes in complex networks. *Scientific reports* 6 (2016), 19307.

[41] Fragkiskos D. Malliaros and Michalis Vazirgiannis. 2013. To stay or not to stay: modeling engagement dynamics in social graphs. In *CIKM*. 469–478. https://doi.org/10.1145/2505515.2505561

[42] David W. Matula and Leland L. Beck. 1983. Smallest-Last Ordering and clustering and Graph Coloring Algorithms. *J. ACM* 30, 3 (1983), 417–427. https://doi.org/10.1145/2402.322385

[43] Alberto Montresor, Francesco De Pellegrini, and Daniele Miorandi. 2013. Distributed k-Core Decomposition. *IEEE TPDS* 24, 2 (2013), 288–300. https://doi.org/10.1109/TPDS.2012.124

[44] Flaviano Morone, Gino Del Ferraro, and Hernán A Makse. 2019. The k-core as a predictor of structural collapse in mutualistic ecosystems. *Nature Physics* 15, 1 (2019), 95.

[45] Jian Pei, Daxin Jiang, and Aidong Zhang. 2005. On mining cross-graph quasi-cliques. In *KDD*. 228–238. https://doi.org/10.1145/1081870.1081898

[46] Lu Qin, Jeffrey Xu Yu, Lijun Chang, Hong Cheng, Chengqi Zhang, and Xuemin Lin. 2014. Scalable big graph processing in mapreduce. In *SIGMOD*. 827–838. https://doi.org/10.1145/2588555.2593661

[47] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. http://networkrepository.com

[48] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2017. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing. *Proc. VLDB Endow.* 11, 4 (2017), 420–431. https://doi.org/10.1145/3186728.3164139

[49] Ahmet Erdem Sariyüce, Bugra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V. Çatalyürek. 2013. Streaming Algorithms for k-core Decomposition. *PVLDB* 6, 6 (2013), 433–444. https://doi.org/10.14778/2536336.2536344

[50] Ahmet Erdem Sariyüce, Bugra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V. Çatalyürek. 2013. Streaming Algorithms for k-core Decomposition. *PVLDB* 6, 6 (2013), 433–444. https://doi.org/10.14778/2536336.2536344

[51] Ahmet Erdem Sariyüce and Ali Pinar. 2016. Fast Hierarchy Construction for Dense Subgraphs. *PVLDB* 10, 3 (2016), 97–108. https://doi.org/10.14778/3021924.3021927

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

[53] Yingxia Shao, Lei Chen, and Bin Cui. 2014. Efficient cohesive subgraphs detection in parallel. In *SIGMOD*. 613–624. https://doi.org/10.1145/2588555.2593665

[54] Kijung Shin, Tina Eliassi-Rad, and Christos Faloutsos. 2016. CoreScope: Graph Mining Using k-Core Analysis - Patterns, Anomalies and Algorithms. In *ICDM*. 469–478. https://doi.org/10.1109/ICDM.2016.0058

[55] Jia Wang and James Cheng. 2012. Truss Decomposition in Massive Networks. *PVLDB* 5, 9 (2012), 812–823. https://doi.org/10.14778/2311906.2311909

[56] Kai Wang, Xin Cao, Xuemin Lin, Wenjie Zhang, and Lu Qin. 2018. Efficient Computing of Radius-Bounded k-Cores. In *ICDE*. 233–244. https://doi.org/10.1109/ICDE.2018.00030

[57] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. 2020. Efficient Bitruss Decomposition for Large-scale Bipartite Graphs. In *ICDE*. IEEE, 661–672. https://doi.org/10.1109/ICDE48307.2020.00063

[58] Kai Wang, Wenjie Zhang, Xuemin Lin, Ying Zhang, Lu Qin, and Yuting Zhang. 2021. Efficient and Effective Community Search on Large-scale Bipartite Graphs. In *ICDE*.

[59] Yue Wang, Xun Jian, Zhenhua Yang, and Jia Li. 2017. Query Optimal k-Plex Based Community in Graphs. *DSE* 2, 4 (2017), 257–273. https://doi.org/10.1007/s41019-017-0051-3

[60] 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. https://doi.org/10.1109/ICDE.2016.7498235

[61] Huanhuan Wu, James Cheng, Yi Lu, Yiping Ke, Yuzhen Huang, Da Yan, and Hejun Wu. 2015. Core decomposition in large temporal graphs. In *2015 IEEE International Conference on Big Data*. IEEE Computer Society, 649–658. https://doi.org/10.1109/BigData.2015.7363809

[62] Long Yuan, Lu Qin, Xuemin Lin, Lijun Chang, and Wenjie Zhang. 2017. I/O efficient ECC graph decomposition via graph reduction. *VLDB J.* 26, 2 (2017), 275–300. https://doi.org/10.1007/s00778-016-0451-4

[63] Chen Zhang, Fan Zhang, Wenjie Zhang, Boge Liu, Ying Zhang, Lu Qin, and Xuemin Lin. 2020. Exploring Finer Granularity within the Cores: Efficient (k, p)-Core Computation. In *ICDE*. IEEE, 181–192. https://doi.org/10.1109/ICDE48307.2020.00023

[64] Fan Zhang, Wenjie Zhang, Ying Zhang, Lu Qin, and Xuemin Lin. 2017. OLAK: An Efficient Algorithm to Prevent Unraveling in Social Networks. *PVLDB* 10, 6 (2017), 649–660. https://doi.org/10.14778/3055330.3055332

[65] Fan Zhang, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2017. When Engagement Meets Similarity: Efficient (k, r)-Core Computation on Social Networks. *PVLDB* 10, 10 (2017), 998–1009. https://doi.org/10.14778/3115404.3115406

[66] Yikai Zhang, Jeffrey Xu Yu, Ying Zhang, and Lu Qin. 2017. A Fast Order-Based Approach for Core Maintenance. In *ICDE*. IEEE Computer Society, 337–348. https://doi.org/10.1109/ICDE.2017.93

[67] Feng Zhao and Anthony K. H. Tung. 2012. Large Scale Cohesive Subgraphs Discovery for Social Network Visual Analysis. *PVLDB* 6, 2 (2012), 85–96. https://doi.org/10.14778/2535568.2448942

[68] Rui Zhou, Chengfei Liu, Jeffrey Xu Yu, Weifa Liang, Baichen Chen, and Jianxin Li. 2012. Finding maximal k-edge-connected subgraphs from a large graph. In *EDBT*. 480–491. https://doi.org/10.1145/2247596.2247652

[69] Andy Diwen Zhu, Wenqing Lin, Sibo Wang, and Xiaokui Xiao. 2014. Reachability queries on large dynamic graphs: a total order approach. In *SIGMOD*. 1323–1334. https://doi.org/10.1145/2588555.2612181

[70] Rong Zhu, Zhaonian Zou, and Jianzhong Li. 2018. Diversified Coherent Core Search on Multi-Layer Graphs. In *ICDE*. 701–712. https://doi.org/10.1109/ICDE.2018.00069