A Fast Order-Based Approach for Core Maintenance

Yikai Zhang[†], Jeffrey Xu Yu[†], Ying Zhang[‡], and Lu Qin[‡]

[†]*The Chinese University of Hong Kong, Hong Kong, China*
[‡]*Centre for QCIS, FEIT, University of Technology, Sydney, Australia*
{ykzhang,yu}@se.cuhk.edu.hk; {Ying.Zhang,Lu.Qin}@uts.edu.au

Abstract—Graphs have been widely used in many applications such as social networks, collaboration networks, and biological networks. One important graph analytics is to explore cohesive subgraphs in a large graph. Among several cohesive subgraphs studied, k-core is one that can be computed in linear time for a static graph. Since graphs are evolving in real applications, in this paper, we study core maintenance which is to reduce the computational cost to compute k-cores for a graph when graphs are updated from time to time dynamically. We identify drawbacks of the existing efficient algorithm, which needs a large search space to find the vertices that need to be updated, and has high overhead to maintain the index built, when a graph is updated. We propose a new order-based approach to maintain an order, called k-order, among vertices, while a graph is updated. Our new algorithm can significantly outperform the state-of-the-art algorithm up to 3 orders of magnitude for the 11 large real graphs tested. We report our findings in this paper.

I. INTRODUCTION

Due to the ubiquity of graph data in different applications, graph analytics has attracted much attention from both research and industry communities. One major issue in graph analytics is to identify cohesive subgraphs in the graphs, such as k-cores [4], k-trusses [15], cliques, n-cliques, and n-clans [6]. Among them, a k-core is defined as the maximal subgraph of an undirected graph G such that all vertices in the subgraph have degree of at least k. For each vertex v in G, the core number of v is defined as the maximum k such that v is contained in the k-core of G. k-cores can be computed in linear time, whereas the time complexity for k-trusses is $\mathcal{O}(m^{1.5})$ [15], and cliques, n-cliques, and n-clans are NP-hard problems [6]. Due to its linear computability, k-cores have been widely used in many real-world applications, including graph visualization [3], community search [11], system structure analysis [17], network topology analysis [2] and so on. There are many works studying how to compute the core number for every vertex in a static graph efficiently. Such a problem is known as a *core decomposition* problem [4], [5], [10], [12], [16].

However, in many real-world applications, such as online social network, collaboration network, and Internet, graphs are evolving where vertices/edges will be inserted/removed over time dynamically. There are increasing interests to study how to handle dynamic graphs. To mention a few, [19] proposes a general technique to maintain a class of 2-hop labels for reachability queries in dynamic graphs; [8] presents a scalable algorithm to maintain approximate densest subgraphs in the dynamic graph model with provable guarantee; [9] solves the problem of incrementally maintaining the matches to a query pattern when updates to the data graph are allowed. The problem of maintaining core numbers for an evolving graph is called *core maintenance*. In brief, after inserting an edge into
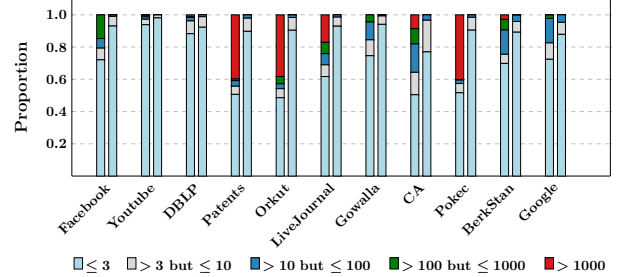

Fig. 1: The distribution of the number of vertices visited: the traversal insertion algorithm (left bar) vs our new order-based insertion algorithm (right bar)

or removing an edge from an undirected graph $G = (V, E)$, the problem is how to update the core numbers for the vertices that need to be updated. There are two important tasks in a core maintenance algorithm. First, it needs to identify the set of vertices whose core numbers need to be updated when an edge is inserted/removed. Such a set is denoted as V^*. Second, it recomputes the core numbers for the vertices in V^*. In order to achieve high efficiency, an index is used and the index needs to be maintained accordingly. In this paper, we study efficient in-memory core maintenance. Ideally, a good core maintenance algorithm should satisfy the following three criteria: (a) a small performance variation among edge updates, (b) a small cost for identifying V^*, and (c) a small cost for updating the index.

Sariyüce et al. in [13] propose an algorithm, called traversal algorithm, which is the state-of-the-art approach for core maintenance. The traversal algorithm searches for V^* only in a local region near the edge inserted/removed. Therefore, it is much faster than recomputing core numbers from scratch. However, the traversal algorithm has drawbacks. First, it shows high variation in terms of performance when edges are inserted into a graph. We conduct testings to insert 100,000 edges into 11 real graphs tested (Table I). For each edge inserted, the algorithm visits some vertices, denoted as V', in order to identify V^*. We show the distribution of the size of V' over all the 100,000 edges inserted in Fig. 1. As shown in Fig. 1, for each of the 11 real graphs tested, there are two bars. The left bar is by the traversal insertion algorithm, whereas the right bar is by our new order-based insertion algorithm. A bar shows the corresponding proportion of edges inserted for different $|V'|$. For example, for a large proportion of the 100,000 edges inserted, both our algorithm and the traversal algorithm only visit up to 3 vertices. However, the traversal insertion algorithm needs to visit more than 1,000 vertices for a non-small proportion of edge insertions, as shown by certain left bars. Our order-based algorithm only needs to visit up to 100 vertices for any of the 11 graphs, showing small performance
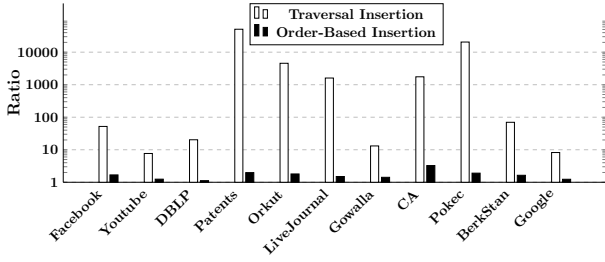
Fig. 2: Ratio of number of vertices visited: the traversal insertion algorithm vs our new order-based insertion algorithm

variation. Second, we show the ratio of the size of the set of vertices visited ($|V'|$) to the size of the set of vertices ($|V^*|$) that need to be updated, i.e., $\frac{\text{sum of } |V'| \text{ over all insertions}}{\text{sum of } |V^*| \text{ over all insertions}}$ for the 100,000 edge insertions over each of the same 11 graphs in Fig. 2. For the traversal insertion algorithm, the ratio is at least 7 and can be up to 10,000 for Patents and Pokec. It suggests the deficiency of the traversal insertion algorithm, even though it is the state-of-the-art. In contrast, for our new insertion algorithm, the ratios are below 4 for all graphs tested and can be down to about 1. Third, for the traversal algorithm, the cost to maintain the index is high. The traversal algorithm maintains a value pcd(v), called the pure-core degree of v, for each v. Here, pcd(v) is computed from the core numbers of all vertices in the 2-hops neighborhood of v. Therefore, if the core number of a vertex changes, a large number of vertices may have their pcd values to be updated, which incurs high cost. We will revisit the traversal algorithm in details in Section IV.

The main contributions of this work are summarized below. First, we identify the deficiency of the state-of-the-art traversal algorithm [13] for core maintenance. Second, we propose new order-based in-memory algorithms for core maintenance: OrderInsert for edge insertion and OrderRemoval for edge removal. Note that, in this work, we do not consider vertex insertion/removal, since they can be simulated by a sequence of edge insertions and removals [11], [13], [16]. The main idea behind our new order-based algorithms is that we explicitly maintain a k-order among vertices such that $u \preceq v$ for every two vertices in a graph G. Here, a k-order, $(v_1, v_2, \ldots, v_n)$, for every vertex v_i in a graph G, is an instance of all the possible vertex sequences produced by a core decomposition algorithm [4]. In other words, there are many possible vertex sequences produced by a core decomposition algorithm, and we ensure the k-order under maintenance is one of them at any time for any evolving graph G. The transitivity holds. That is, $v_h \preceq v_j$ if $v_h \preceq v_i$ and $v_i \preceq v_j$. Given a k-order maintained for G, when an edge is inserted/removed, we maintain the k-order for the new G with the edge inserted/removed. Third, we have conducted extensive performance studies, and show that we can achieve high efficiency up to 2,083 times faster than the traversal insertion algorithm.

The rest of this paper is organized as follows. We discuss the related work in Section II. We give the preliminaries in Section III, and review the traversal insertion/removal algorithm in Section IV. We discuss our new order-based algorithm, namely, OrderInsert and OrderRemoval, in Section V, and give some implementation details in Section VI. We conduct extensive performance studies, and report the results in Section VII, and conclude this work in Section VIII.

II. RELATED WORK

Identifying cohesive subgraphs is an important problem in graph analytics. Common cohesive subgraphs are clique (maximal clique), k-plex, n-clique, n-clan [6], k-truss [15], k-core [4] and so on, among which k-core is the only one known to have a linear algorithm. We review the related work on computing k-core (core decomposition) and maintaining k-core (core maintenance) respectively below.

Core Decomposition: The core decomposition is to efficiently compute for each vertex its core number. [4] proposes an in-memory linear algorithm ($\mathcal{O}(m + n)$). Their algorithm uses a bottom-up process in which k-cores are computed in the order of 1-core, 2-core, 3-core, $\cdots$. To process graphs that can not reside in the memory, [5] proposes an external algorithm, which runs in a top-down manner such that the whole graph does not need to be loaded to memory to compute higher cores. [16] proposes a new semi-external algorithm, which assumes that the memory has at least $\Omega(n)$ size and can maintain a small constant amount of information for each vertex in memory. [12] considers core decomposition under a distributed setting. [10] investigates core decomposition of large graphs using GraphChi, WebGraph, and external model, and compares their performance in a single PC.

Core Maintenance: Given a large graph, it takes high computational cost to recompute core numbers for vertices when some edges are inserted/removed, even though there exist linear algorithms. The core maintenance is to maintain core numbers efficiently when edges and vertices are inserted/removed. Since the insertions and removals of vertices can be simulated by a sequence of edge insertions and removals, all existing works [11], [13], [16], [1] consider only edge updates. The key issue is how to efficiently identify the set of vertices, denoted as V^*, whose core numbers need to be updated, around the edge inserted/removed. Recently, [11] and [13] independently found that, when an edge is inserted or removed, the induced subgraph of V^* is connected and moreover, V^* resides in the subcore which the edge is in. Based on this observation, [13] proposes algorithms that are linear in the size of the subcore, whereas [11] proposes a quadratic algorithm. Among the algorithms proposed in [13], the traversal algorithm is the state-of-the-art for core maintenance. We will revisit the traversal algorithm later, and show that we can significantly improve the efficiency when a graph can be held in memory. A semi-external algorithm for core maintenance is proposed in [16] to reduce I/O cost; but it is not optimized for CPU time. The algorithm proposed by [1] is similar to the SubCore algorithm of [13] but is less efficient due to weaker bounds.

III. THE PRELIMINARIES

Let $G = (V, E)$ be an undirected graph, where $V(G)$ denotes the set of vertices and $E(G)$ represents the set of edges in G. We denote the number of vertices and edges of G by n and m respectively. In this paper, we use nbr(u, G) to denote the set of neighbors of a vertex $u \in V(G)$, i.e., nbr(u, G) = $\{v \in V(G) : (u, v) \in E(G)\}$. Besides, we define the degree of u in G as deg(u, G) = $|\text{nbr}(u, G)|$. When the context is clear, we will use nbr(u) and deg(u) instead of nbr(u, G) and deg(u, G). We say a graph G' is a subgraph of G, denoted as $G' \subseteq G$, if $V(G') \subseteq V(G)$ and
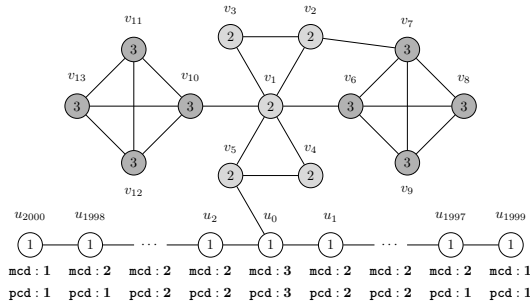
Fig. 3: A Sample Graph G.

$E(G') \subseteq E(G)$. Given a subset $V' \subseteq V$, the subgraph induced by V', denoted as $G(V')$, is defined as $G(V') = (V', E')$ where $E' = \{(u, v) \in E : u, v \in V'\}$.

A subgraph G_k of G is called a **k-core** if it satisfies the following conditions: (1) for $\forall u \in V(G_k)$, $\deg(u, G_k) \geq k$; (2) G_k is maximal. $G_k = \emptyset$ if the k-core of G does not exist. For a given k, the k-core G_k of a graph G is unique. Moreover, $G_{k+1} \subseteq G_k$, for $\forall k \geq 0$. Note that when $k = 0$, G_0 is just G. A closely related concept to k-core is **core number**, which is defined as follows: For each vertex $u \in V(G)$, its core number $\mathsf{core}(u, G)$ is defined as $\mathsf{core}(u, G) = \max\{k : u \in V(G_k)\}$. When the context is clear, for simplicity, we use $\mathsf{core}(u)$ to denote the core number of u instead. Given a graph $G = (V, E)$, a set of vertices $\mathcal{SC} \subseteq V$ is called a **k-subcore** if (1) $\forall u \in \mathcal{SC}$, $\mathsf{core}(u) = k$; (2) the induced subgraph $G(\mathcal{SC})$ is connected; (3) $\mathcal{SC}$ is maximal. For a vertex u, the subcore containing u is denoted as $\mathsf{sc}(u)$.

Example 3.1: Consider the graph G in Fig. 3. The subgraph induced by $\{v_6, v_7, \cdots, v_{13}\}$ is the 3-core of G because in the induced graph, every vertex has a degree at least 3. There does not exist a 4-core in G. We have $\mathsf{core}(v_i) = 3$ for $6 \leq i \leq 13$, $\mathsf{core}(v_i) = 2$ for $1 \leq i \leq 5$ and $\mathsf{core}(u_i) = 1$ for $0 \leq i \leq 2000$. Also, $\{v_1, v_2, v_3, v_4, v_5\}$ and $\{u_i\}$ are the unique 2-subcore and 1-subcore of G, respectively. There are two 3-subcores in G by $\{v_6, v_7, v_8, v_9\}$ and $\{v_{10}, v_{11}, v_{12}, v_{13}\}$.

Core Decomposition: Given a graph G, the problem of computing the core number for each $u \in V(G)$ is called core decomposition. [4] proposed an $\mathcal{O}(m + n)$ algorithm, denoted as CoreDecomp, whose sketch is shown in Algorithm 1. The general idea is as follows: To compute the k-core G_k of G, it repeatedly remove those vertices (and their adjacent edges) whose degrees are less than k. When there is no more vertex to remove, the resulting graph is the k-core of G.

Core Maintenance: The problem of core maintenance is to maintain the k-core by maintaining the core numbers, when edges are inserted to or removed from G. This is because it is known that k-core G_k can be efficiently computed using the core numbers of the vertices, since a vertex $u \in V(G_k)$ if and only if $\mathsf{core}(u) \geq k$. In addition, an index can be constructed for core numbers of the vertices. From now on, we use the notation V^* to denote the set of vertices whose core numbers need to be updated after inserting or removing an edge. The insertion or removal of vertices can be simulated as a sequence of edge insertions and removals. Hence in this paper, we focus on efficient core maintenance under edge updates.

Algorithm 1: CoreDecomp (G)

1   $k \leftarrow 1$;
2   **while** G *is not empty* **do**
3   **while** $\exists u \in V(G)$ *such that* $\deg(u) < k$ **do**
4   let u be a vertex with $\deg(u) < k$;
5       $\deg(w) \leftarrow \deg(w) - 1$ for each $w \in \mathsf{nbr}(u)$;
6   remove u and its adjacent edges from G;
7       $\mathsf{core}(u) \leftarrow k - 1$;
8     $k \leftarrow k + 1$;
9   **return** core;

Here we present two theorems given in [11], [13] based on which the correctness of our algorithms is proved.

Theorem 3.1: [11], [13] *After inserting to (resp. removing from) $G = (V, E)$ an edge, the core number of a vertex $u \in V$ increases (resp. decreases) by at most 1.*

Theorem 3.2: [11], [13] *Suppose an edge (u, v) with $K = \mathsf{core}(u) \leq \mathsf{core}(v)$ is inserted to (resp. removed from) G. Suppose V^* is non-empty. We have the following: (1) if $\mathsf{core}(u) < \mathsf{core}(v)$, then $u \in V^*$ and $V^* \subseteq \mathsf{sc}(u)$; (2) if $\mathsf{core}(u) = \mathsf{core}(v)$, then both u and v are in V^* (resp. at least one of u and v is in V^*) and $V^* \subseteq \mathsf{sc}(u) \cup \mathsf{sc}(v)$; (3) the induced subgraph of V^* in $G \cup \{(u, v)\}$ (resp. G) is connected.*

Theorem 3.1 suggests that we only need to find V^*, the set of vertices whose core numbers need to be updated. Once V^* is found, we can then increase (or decrease) core for vertices in V^* by 1 accordingly. Theorem 3.2 suggests two things. (1) Only vertices w with $\mathsf{core}(w) = K$ *may* be in V^*, i.e., $\mathsf{core}(w)$ may need to be updated. (2) We can search for V^* in a small local region near the inserted (or removed) edge (i.e., the subcores containing u and v), rather than in the whole G.

IV. THE TRAVERSAL ALGORITHM

In this section, we discuss the state-of-the-art core maintenance solution, called the traversal algorithm [13], which is designed on top of two important notions, namely $\mathsf{mcd}(u)$ and $\mathsf{pcd}(u)$. We introduce them below.

The **max-core degree** of a vertex u, denoted as $\mathsf{mcd}(u)$, is defined as the number of u's neighbors, w, such that $\mathsf{core}(w) \geq \mathsf{core}(u)$. Intuitively, $\mathsf{mcd}(u)$ counts u's neighbors in the $(\mathsf{core}(u))$-core. By the definition of k-core, $\mathsf{mcd}(u) \geq \mathsf{core}(u)$. The **pure-core degree** of a vertex u, denoted as $\mathsf{pcd}(u)$, is defined as the number of u's neighbors, w, such that either $\mathsf{core}(w) = \mathsf{core}(u)$ and $\mathsf{mcd}(w) > \mathsf{core}(w)$ or $\mathsf{core}(w) > \mathsf{core}(u)$. The main difference between pcd and mcd is that $\mathsf{pcd}(u)$ further excludes neighbors w with $\mathsf{core}(w) = \mathsf{core}(u)$ and $\mathsf{mcd}(w) = \mathsf{core}(w)$. Thus $\mathsf{pcd}(u) \leq \mathsf{mcd}(u)$. We show $\mathsf{mcd}(u_i)$ and $\mathsf{pcd}(u_i)$, for $\{u_i\}$, in Fig. 3 and an example below which demonstrates why mcd and pcd are useful.

Example 4.1: Suppose edge (v_4, u_0) is inserted to G in Fig. 3. Due to the newly inserted edge, both $\mathsf{mcd}(u_0)$ and $\mathsf{pcd}(u_0)$ become 4. By Theorem 3.2, only core numbers of vertices in $\{u_i\}$ may need to be updated. We show two cases. First, consider u_{1999}. u_{1999} can not be in the 2-core of $G \cup \{(v_4, u_0)\}$, because the number of its neighbors in the new 2-core is upper bounded by $\mathsf{mcd}(u_{1999}) < 2$. To see why $\mathsf{mcd}(u_{1999})$ is an upper bound, recall that by Theorem 3.1 and Theorem 3.2,
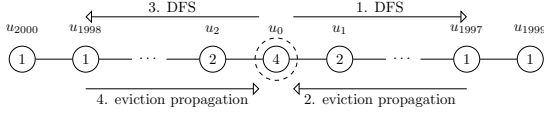
Fig. 4: Illustration of Example 4.2

only vertices with core numbers greater than or equal to 1 may be in the new 2-core. Second, consider u_{1997}. Regarding $\mathsf{mcd}(u_{1997})$, the max possible number of u_{1997}'s neighbors in the new 2-core is $\mathsf{mcd}(u_{1997}) = 2$, so $\mathsf{core}(u_{1997})$ *may* need to be updated. Regarding $\mathsf{pcd}(u_{1997})$, the max possible number of u_{1997}'s neighbors in the new 2-core is $\mathsf{pcd}(u_{1997}) = 1 < 2$. $\mathsf{pcd}(u_{1997})$ does not count u_{1999}, which can not be in the new 2-core because $\mathsf{mcd}(u_{1999}) = \mathsf{core}(u_{1999}) = 1$. Hence, with $\mathsf{pcd}(u_{1997})$, there is no need to update $\mathsf{core}(u_{1997})$.

A. The Traversal Insertion Algorithm

The insertion algorithm employs an expand-shrink framework to determine V^*, the set of vertices whose core numbers need to be updated. It works as follows: When a new edge (u, v) with $K = \mathsf{core}(u) \leq \mathsf{core}(v)$ is inserted into G, it first updates mcd and pcd accordingly based on the old core values. Because $\mathsf{core}(u) \leq \mathsf{core}(v)$, it selects u as the root. Based on Theorem 3.2 and the discussion above, in order to find V^*, it issues a DFS starting from the root and only visits w with $\mathsf{core}(w) = K$ and $\mathsf{mcd}(w) > K$. The algorithm maintains a value $\mathsf{cd}(w)$ for every vertex w visited, where $\mathsf{cd}(w)$ represents the max possible number of w's neighbors in the new $(K + 1)$-core. Initially $\mathsf{cd}(w) = \mathsf{pcd}(w)$. During the DFS, it stops the search when the vertex w visited is confirmed to be $w \notin V^*$ (i.e., $\mathsf{cd}(w) \leq K$). It evicts w (i.e., mark $w \notin V^*$) and decreases the cd values of its neighbors by 1. In addition, it conducts a backward search from w to find more visited vertices to evict. Such a process is called an eviction propagation. After the eviction propagation ends, it will continue DFS in the original order, skipping w. When the algorithm terminates, it is guaranteed that all visited but not evicted vertices constitute V^*. As the final step, it needs to update mcd and pcd for later insertions and removals.

Example 4.2: Suppose (v_4, u_0) is inserted to the graph G in Fig. 3. The initial cd values for all u_i are shown in Fig. 4. The traversal insertion algorithm selects u_0 as the root and issues a DFS from u_0. The DFS recursively visits those vertices u_i with $\mathsf{core}(u_i) = 1$ and $\mathsf{mcd}(u_i) > 1$. Without loss of generality, assume the DFS first traverses the right part of u_0. When DFS reaches u_{1997}, it finds $\mathsf{cd}(u_{1997}) \leq 1$, thus $\mathsf{core}(u_{1997})$ does not need to be updated and u_{1997} is evicted. Further, the eviction propagates back to u_1 as follows: Due to eviction of u_{1997}, $\mathsf{cd}(u_{1995})$ is decreased by 1 and becomes 1. Thus u_{1995} is also evicted. This process continues and finally it evicts all $u_{1997}, \cdots, u_1$. The DFS then traverses the left part. Similarly, $u_2, \cdots, u_{1998}$ are all evicted. At last, u_0 is the only vertex that is visited but not evicted and therefore $V^* = \{u_0\}$.

In Example 4.2, although u_0 is the only vertex whose core number needs to be updated, the traversal insertion algorithm needs to visit much more (specifically, 1,999) vertices to determine $V^* = \{u_0\}$. As shown in Fig. 1 and Fig. 2, such an issue is not rare and hurts the performance significantly.
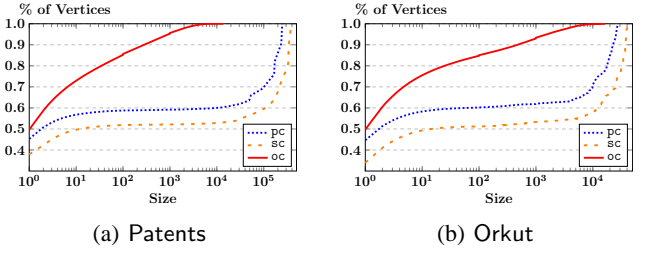

(a) Patents

(b) Orkut

Fig. 5: Cumulative size distribution of pc, sc, and oc.

Such a high overhead is due to the potential large search space used by the algorithm. Let V' be the search space, i.e., the set of vertices that the traversal insertion algorithm visits, in order to obtain V^*. In order to show an upper bound of $|V'|$, we introduce a concept called *pure-core* [13], in Definition 4.1.

Definition 4.1: For each $u \in V(G)$, its **pure-core**, denoted as $\mathsf{pc}(u)$, is defined as $\mathsf{pc}(u) = \{u\} \cup \mathcal{PC}$, where $\mathcal{PC}$ is a set of vertices that satisfies the following 3 conditions: (1) for $\forall w \in \mathcal{PC}$, $\mathsf{mcd}(w) > \mathsf{core}(w)$ and $\mathsf{core}(w) = \mathsf{core}(u)$; (2) $G(\{u\} \cup \mathcal{PC})$ is connected; and (3) $\mathcal{PC}$ is maximal.

It is not hard to observe that $V' \subseteq \mathsf{pc}(u) \cup \mathsf{pc}(v)$ when an edge (u, v) is inserted; that is, the size of pure-cores decides the max number of vertices that are visited by the traversal insertion algorithm. We show the cumulative distribution of pc, for two of the largest graphs tested (Table I), namely, Patents and Orkut, in Fig. 5. Fig. 5 shows the % of vertices whose pure-cores (pc) are smaller than a certain size s for all possible s in the x-axis. We observe that pc is with high variance in distribution. Moreover, many vertices have a very large pure-core size. For example, for both graphs, more than 10% of vertices have their pure-core size greater than 10,000, which is very high and leads to high overhead.

B. The Traversal Removal Algorithm

The removal algorithm first updates mcd and pcd when an edge (u, v) with $K = \mathsf{core}(u) \leq \mathsf{core}(v)$ is removed from G. The algorithm uses the similar idea as shown in the CoreDecomp algorithm (Algorithm 1) to find V^*. It works as follows. Let u be the root (if $\mathsf{core}(v) = K$, v should also be a root). Starting from the root(s), the algorithm finds V^* by repeatedly removing vertices w with $\mathsf{core}(w) = K$ and $\mathsf{cd}(w) < K$ and appending w to V^*, where $\mathsf{cd}(w)$ is initialized as $\mathsf{mcd}(w)$ and counts the max possible # of neighbors of w in the new K-core. When w is removed, the algorithm decreases $\mathsf{core}(w)$ by 1. Since $\mathsf{core}(w)$ becomes $K - 1$, for $z \in \mathsf{nbr}(w)$ with $\mathsf{core}(z) = K$, $\mathsf{cd}(z)$ is decreased by 1. Accordingly, if $\mathsf{cd}(z)$ becomes smaller than K, z is also removed. Such a process continues until no more vertex can be removed and requires only $\mathcal{O}(\sum_{v \in V^*} \deg(v))$ time. At last, mcd and pcd need to be updated for later insertions and removals.

It is important to note that pcd needs to be maintained for the following possible edge insertions, even though pcd is not required in the traversal removal algorithm. The performance of the algorithm is completely dominated by the maintenance of pcd. By the definition, $\mathsf{pcd}(u)$ may be affected by a vertex w that is 2 hops away from u. If $\mathsf{core}(u)$ is decreased by 1 for every $u \in V^*$, in the worst case, $\mathcal{O}(\sum_{v \in \mathsf{nbr}(V^*)} \deg(v))$ vertices may have their pcd affected.

Both the traversal insertion algorithm and the traversal removal algorithm suffer from the same performance problem to maintain pcd, and the cost to maintain pcd is high. In some cases, although the algorithm visits only a small number of vertices to find V^*, the benefit brought however is neutralized by the cost to maintain pcd.

V. AN ORDER-BASED ALGORITHM

There are two main issues in the traversal algorithm. One is the cost of finding V^* and the other is the cost of maintaining pcd. The issue of finding V^* by the traversal insertion algorithm is more serious than the traversal removal algorithm. Unlike the traversal removal algorithm, the traversal insertion algorithm does not take the idea shown in CoreDecomp (Algorithm 1). Recall the idea shown in CoreDecomp. When a new edge (u, v) with $K = \mathsf{core}(u) \leq \mathsf{core}(v)$ is inserted, it identifies V^* by repeatedly removing vertices whose degrees are less than $K + 1$. Then, V^* is identified as the set of the vertices taken from the vertices remained if their original core numbers are K. The main reason that the traversal insertion algorithm does not do so is due to the fact that the cost of core maintenance otherwise will become as high as the cost of core decomposition.

In this work, we revisit the idea presented in CoreDecomp (Algorithm 1), and design new algorithms to significantly improve the performance for core maintenance for both edge insertions and removals. The challenges by taking the idea behind CoreDecomp for core maintenance are twofold. First, we need to find an efficient way to maintain the degrees of the neighbors when a vertex is removed during core maintenance, instead of removing vertices as done by CoreDecomp. In other words, we need to maintain the order used by CoreDecomp, but we cannot afford to compute the order in core maintenance. Second, because the number of vertices to be affected can be large, it is impractical to remove vertices online one-by-one. The solution we propose to address the challenges is to introduce an order, called k-order, with which we can reduce the core maintenance cost. The proofs for our lemmas and theorems can be found in the full version [18].

Consider a graph G. Let $G^0 = G$ and G^i be the graph after i insertion/removal of edges. The k-order is initially defined over G^0, which is the order by CoreDecomp (Algorithm 1). In details, the k-order, denoted as $\preceq$, is that $u \preceq v$ if and only if vertex u will be removed in Algorithm 1 before v. It is obvious that $u \preceq v$ if $\mathsf{core}(u) < \mathsf{core}(v)$. It is worth mentioning that either $u \preceq v$ or $v \preceq u$ is possible when $\mathsf{core}(v) = \mathsf{core}(u)$. In such a case, one of the two can be used, if it can be obtained by Algorithm 1. When $\preceq$ is determined for G^0, we will maintain the order $\preceq$ such that it is a k-order for graph G^i for $i > 0$. In the following, for simplicity, we use G instead of G^i and G' instead of G^{i+1} when the context is clear.

Definition 5.1: (**k-order**) Given a graph G, let G^0 be G. Assume G^i is a graph after i insertion/removal of edges. The k-order $\preceq$ is defined for any u and v over G^i as follows.

- When $\mathsf{core}(u) < \mathsf{core}(v)$, $u \preceq v$.
- When $\mathsf{core}(u) = \mathsf{core}(v)$, $u \preceq v$ if u is removed before v by CoreDecomp (Algorithm 1) for G^i.
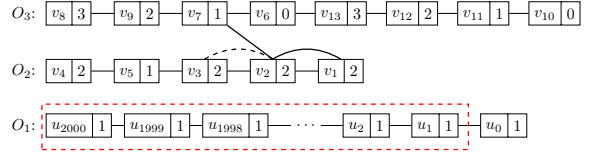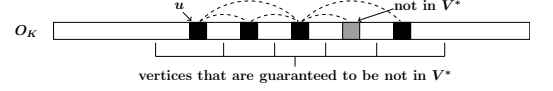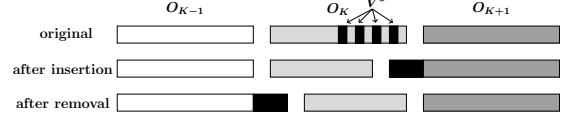

Fig. 6: The k-order for G in Fig. 3

(a) Identification of V^* in the Insertion Algorithm

(b) Maintenance of k-Orders

Fig. 7: Algorithm Framework

A k-order, $(v_1, v_2, \ldots, v_n)$, for $v_i \in V(G)$, is an instance of all the possible vertex sequences produced by Algorithm 1. The transitivity holds; that is, $u \preceq v$ if $u \preceq w$ and $w \preceq v$.

Definition 5.2: (**remaining degree**) For a vertex u in G^i, the remaining degree of u, denoted as $\deg^+(u)$, is defined as: $\deg^+(u) = |\{v \in \mathsf{nbr}(u) : u \preceq v\}|$.

Here, $\deg^+(u)$ is the degree of the neighbors that appear after u. Given k-order $\preceq$ defined, we use O_k to denote the sequence of vertices in k-order whose core numbers are k. It is obvious that for a vertex u in O_k, $\deg^+(u) \leq k$. And we have a sequence of $O_0 O_1 O_2 \cdots$, where $O_i \preceq O_j$ if $i < j$. It is clear that $\preceq$ defined over the sequence of $O_0 O_1 O_2 \cdots$ is implied by the k-order ($\preceq$).

Lemma 5.1: *Given a graph G, the order ($\preceq$) on $O_0 O_1 O_2 \cdots$ over $V(G)$ is the k-order if and only if $\deg^+(v) \leq k$ for every vertex v in O_k, for $\forall k$.*

By Lemma 5.1, in order to maintain the k-order, we maintain every O_k such that $\forall v \in O_k$, $\deg^+(v) \leq k$.

Example 5.1: In Fig. 6, we show a k-order for the graph G (Fig. 3) by showing O_1, O_2, O_3. In O_1, $u_i \preceq u_j$ if $i > j$. In O_2, $v_4 \preceq v_5 \preceq v_3 \preceq v_2 \preceq v_1$. In O_3, $v_8 \preceq v_9 \preceq v_7 \preceq v_6 \preceq v_{13} \preceq v_{12} \preceq v_{11} \preceq v_{10}$. The k-order is one of the orders computed by Algorithm 1 for the graph G. Consider the vertex v_2 in G. v_2 has three neighbors: $v_1, v_3,$ and v_7. Here, both v_1 and v_3 are in O_2 and v_7 is in O_3, because $\mathsf{core}(v_1) = \mathsf{core}(v_3) = 2$ and $\mathsf{core}(v_7) = 3$. In terms of k-order, $v_3 \preceq v_2 \preceq v_1 \preceq v_7$. Therefore, $\deg^+(v_2) = 2$. The $\deg^+(v_i)$ for v_i is shown as a number next to v_i in Fig. 6.

A. An Algorithm Overview

We design new algorithms to deal with an edge (u, v) to be inserted into or removed from G. For simplicity, we assume $K = \mathsf{core}(u) \leq \mathsf{core}(v)$, and $u \preceq v$. Here, $u \in O_K$. Let the resulting graph of the insertion/removal be G'. There are two key issues in our algorithms. First, we need to identify V^*, the set of vertices whose core numbers need to be updated,

efficiently using the k-order $\preceq$ of G. Second, we need to maintain the k-order efficiently such that the new order $\preceq'$ is the k-order of G'.

Edge Insertion: Suppose an edge (u, v) is inserted for $u \in O_K$ and $u \preceq v$. V^* can be computed efficiently. We show the main ideas below.

(i1) A vertex w cannot be in V^* if $w \in O_L$ for $L < K$.
(i2) A vertex w cannot be in V^* if $w \in O_L$ for $K < L$.
(i3) A vertex w in O_K cannot be in V^* if $w \preceq u$.
(i4) A vertex w in O_K *may* be in V^* if $u \preceq w$ and there is a path $w_0, w_1, w_2, \cdots, w_t$ such that $w_0 = u$, $w_t = w$, $(w_i, w_{i+1}) \in E$ and $w_i \preceq w_{i+1}$ for $0 \le i < t$.
(i5) A vertex w that satisfies (i4) is not necessarily in V^*.

Items (i1) and (i2) are proved by Theorem 3.2. We further explain other points using Fig. 7a. In O_K, a vertex w can not be in V^* if $w \preceq u$ by (i3). Fig. 7a also shows that the vertices reachable from u by (i4) are possibly in V^*, but not all the vertices reachable from u are included in V^* by (i5).

To compute V^*, we design an efficient algorithm that can "jump" from a candidate of V^* to another following the k-order of G. Therefore, the search space for obtaining V^* is significantly reduced. Suppose V^* is obtained. To obtain the k-order $\preceq'$ of the resulting graph G', as shown in Fig. 7b, the set of vertices in V^* in O_K will be moved to the beginning of O_{K+1}. The k-order by the insertion of (u, v) remains unchanged for all vertices in O_L, for either $K < L$ or $K > L$, in G'. The k-order for the vertices that will be in the same O_K in the resulting graph G' needs small adjustment. It is guaranteed that only vertices visited by our algorithm may have their positions changed. The new order $\preceq'$ is a valid k-order of G'.

Edge Removal: Suppose an edge (u, v) is removed. We adopt the similar idea presented in the traversal removal algorithm to find V^*. However, we maintain the k-order instead of pcd used in the traversal removal algorithm to achieve high efficiency. To obtain the k-order of G', we move the vertices in V^* from O_K to the end of O_{K-1}, without affecting the original vertices in O_{K-1}, as shown in Fig. 7b. The resulting order is guaranteed to be a valid k-order of G'. We present more details when we discuss our removal algorithm.

B. The Order-Insertion Algorithm

Identifying V^*: Suppose an edge (u, v) is inserted for $u \in O_K$ and $u \preceq v$. Since (i1) and (i2) are trivial, we focus on (i3) – (i5). We call the vertex u as a root since $u \preceq v$. Note that by insertion of (u, v), $\deg^+(u)$ is increased by 1 to reflect the insertion of (u, v), while $\deg^+(v)$ remains unchanged. Next, we show in Lemma 5.2 that no vertices in O_K can be in V^* if u does not have sufficient neighbors that appear after u in k-order, i.e., $\deg^+(u) \le K$. This implies that we need to update core numbers of vertices in O_K only if $\deg^+(u) > K$. We also show in Lemma 5.3 that there is no need to update the vertices that appear before u in O_K, when $\deg^+(u) > K$. For the vertices that appear after u in k-order in O_K, we show the cases we need to consider.

Lemma 5.2: *No core number needs to be updated for vertices in O_K, if $\deg^+(u) \le K$ after increasing $\deg^+(u)$ by 1.*

Lemma 5.3: *No vertex w that appears before u in O_K can be in V^*, if $\deg^+(u) > K$ after increasing $\deg^+(u)$ by 1.*

Example 5.2: Reconsider Example 4.2. Suppose edge (v_4, u_0) is inserted to G, where $u_0 \preceq v_4$ in G. Then, $\deg^+(u_0)$ becomes 2. Note that u_0 is in O_1 before the new edge is inserted. By Lemma 5.3, all vertices that occur before u_0 in O_1 (the rectangle in Fig. 6) can not be in V^*. Reconsider u_0, $\deg^+(u_0) = 2$ implies that u_0 has at least two neighbors appearing after u_0 in the k-order, which are v_4 and v_5 in this example. Hence, u_0 will be in the 2-core of the new graph $G \cup \{(v_4, u_0)\}$. We have $V^* = \{u_0\}$. As shown in this example, our approach achieves high efficiency, because we only need to visit 1 vertex (i.e., u_0), while the traversal insertion algorithm needs to visit $1,999$ vertices in total to identify V^*.

In Example 5.2, vertex u_0 is the last vertex in O_1. Generally, when a new edge (u, v) is inserted for $u \in O_K$ and $u \preceq v$, u can be in any position in O_K. There may be vertices appearing after u in O_K. Among all such vertices, we further discuss whether a vertex w with $u \preceq w$ in O_K can be a potential candidate in V^*. We first introduce the concept of candidate degree.

Definition 5.3: (candidate degree) For a vertex w in O_K, the candidate degree of w, denoted as $\deg^*(w)$, is defined as: $\deg^*(w) = |\{w' \in \mathsf{nbr}(w) : w' \preceq w \wedge \mathsf{core}(w') = K \wedge w'$ is a potential candidate of $V^*\}|$.

In order to test whether a vertex w can be a potential candidate in V^*, we use $\deg^*(w) + \deg^+(w)$. Recall that $\deg^+(w)$ (Definition 5.2) counts the number of w's neighbors after w in the k-order, whose core numbers are greater than or equal to K. In other words, $\deg^+(w)$ counts how many w's neighbors after w can be in the new $(K+1)$-core. Therefore, $\deg^+(w) + \deg^*(w)$ upper bounds the number of w's neighbors in the new $(K+1)$-core. Specifically, if $\deg^+(w) + \deg^*(w) \le K$, w does not have sufficient neighbors in the new $(K+1)$-core, thus w cannot be in V^*; if $\deg^+(w) + \deg^*(w) > K$ otherwise, w is a potential candidate.

Initially, $\deg^*(w) = 0$ for each $w \in O_K$ since we have not found any candidate yet. We start from u and visit vertices in O_K following the k-order. For each vertex w being visited, we discuss two cases.

Case-1 ($\deg^*(w) + \deg^+(w) > K$): Since w potentially has more than K neighbors in the new $(K+1)$-core, w may be in V^*, i.e., $\mathsf{core}(w)$ may become $K+1$. We put w in a set V_C, which records all current visited potential candidates. Further, each neighbor w' of w that $w \preceq w'$ in O_K, obtains one candidate degree ($\deg^*(w')$ increased by 1) to reflect the existence of w. In other words, for each such w', the potential number of neighbors in the new $(K+1)$-core increases by 1. Note that u is a special example in this case because $\deg^*(u) = 0$ and $\deg^+(u) > K$. We next visit the vertex that is next to w in the k-order.

Case-2 ($\deg^*(w) + \deg^+(w) \le K$): w does not have sufficient neighbors in the new $(K+1)$-core, since $\deg^*(w) + \deg^+(w) \le K$. Therefore, $w \notin V^*$. There are two subcases, namely, **Case-2a** for $\deg^*(w) = 0$ and **Case-2b** for $\deg^*(w) \ne 0$.
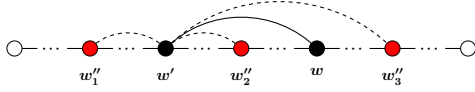
Fig. 8: Demonstration of **Case-2b**.

For **Case-2a**, w cannot be in V^*. We next visit the vertex w'' which is next to w in the k-order. It is possible that w'' again is in **Case-2a**, implying that $w'' \notin V^*$. More generally, let w' be the first vertex after w in O_K such that $\deg^*(w') \neq 0$. For any w'' in the range $[w, w')$ in the k-order, $\deg^*(w'') = 0$. As a result, these w'''s cannot be in V^* and we can skip them, directly "jump" to visit w' next. Specially, if such w' does not exist, all vertices after w cannot be in V^*.

For **Case-2b**, it differs from **Case-2a** for $\deg^*(w) \neq 0$. Here, $\deg^*(w) \neq 0$ implies that some neighbor w' that $w' \preceq w$ is considered as a potential candidate. In other words, $\deg^*(w') + \deg^+(w') > K$ (refer to **Case-1** where w' is in the position of w in **Case-1**). Excluding w from V^* results in $\deg^+(w')$ to be decreased by 1 for each neighbor vertex w' that $w' \preceq w$ and $w' \in V_C$. Recall that V_C is the set of vertices that currently are candidates for V^*. If $\deg^*(w') + \deg^+(w') \leq K$ after such a update, then we can remove w' from V_C since w' is not a candidate any more, which may further result in other potential candidates to be removed from V_C. The situation can be demonstrated using Fig. 8. Suppose w' is not a potential candidate after $\deg^+(w')$ decreases by 1. Then (1) $\deg^+(w''_1)$ is decreased by 1 for each neighbor w''_1 of w' that $w''_1 \in V_C$ and $w''_1 \preceq w'$; (2) $\deg^*(w''_2)$ is decreased by 1 for each neighbor w''_2 of w' that $w''_2 \in V_C$ and $w' \preceq w''_2 \preceq w$; (3) $\deg^*(w''_3)$ is decreased by 1 for each neighbor w''_3 of w' that $w \preceq w''_3$. For w''_1 and w''_2, they may not be a candidate any more after update and will be removed from V_C. The chain effect can further propagate. For w''_3, we just update its $\deg^*$ and it will be processed when we visit it later. Recall that we visit vertices in O_K following the k-order. We choose the vertex next to w in the k-order as the next vertex to visit.

When we reach the end of O_K, $V^* = V_C$. We explain the rationale as follows. For each $w \in V_C$, we have (1) $\deg^*(w) = |\{w' \in \mathrm{nbr}(w) : w' \in V_C \wedge w' \preceq w\}|$, and (2) $\deg^+(w) = |\{w' \in \mathrm{nbr}(w) : w' \in V_C \wedge w \preceq w' \text{ or } \mathrm{core}(w') > K\}|$. Therefore, $\deg^*(w) + \deg^+(w)$ is the number of the neighbors of w that are in $V_{>K} \cup V_C$ and is greater than or equal to $K+1$, where $V_{>K} = \{v \in V : \mathrm{core}(v) > K\}$. Hence, $G(V_{>K} \cup V_C)$ is the new $(K+1)$-core of G'. In other words, $V^* = V_C$.

Maintaining k-order: Since the core numbers of vertices in V^* change from K to $K+1$, we need to obtain two new sequences O'_K and O'_{K+1} for vertices with core number K and $K+1$ respectively, such that the new order $\preceq'$ over $O_0 \cdots O_{K-1} O'_K O'_{K+1} O_{K+2} \cdots$ is a k-order of G'. By Lemma 5.1, we only need to guarantee that (1) $\deg^+(w) \leq K$ for $\forall w \in O'_K$ and (2) $\deg^+(w) \leq K+1$ for $\forall w \in O'_{K+1}$. We discuss O'_K first.

In order to construct O'_K, the main idea is to append a vertex w to O'_K when we encounter a vertex w with $\deg^*(w) + \deg^+(w) \leq K$ during the process of identifying V^*. The rationale is that $\deg^*(w) + \deg^+(w)$ is exactly the number of w's neighbors that have not been appended to O'_K yet or that have core number greater than K. Therefore,

$\deg^*(w) + \deg^+(w)$ is the number of w's neighbors that will appear after w in the new k-order. The new $\deg^+(w)$ is set as $\deg^*(w) + \deg^+(w)$. We give the details. Let w be the vertex currently being visited.

- **Case-1**: w cannot be appended to O'_K temporarily and thus is skipped.
- **Case-2a**: w is appended to O'_K. Because $\deg^*(w) = 0$, $\deg^+(w)$ remains unchanged.
- **Case-2b**: w is appended to O'_K and $\deg^+(w)$ becomes $\deg^*(w) + \deg^+(w)$. For each $w' \in V_C$ that is found $\deg^*(w') + \deg^+(w') \leq K$, we append w' to O'_K and the new $\deg^+(w')$ is set as $\deg^*(w') + \deg^+(w')$.

In order to construct O'_{K+1}, we insert vertices in V_C to the beginning of O_{K+1} such that $\forall w_1, w_2 \in V_C$, $w_1 \preceq' w_2$ in the new order $\preceq'$ if $w_1 \preceq w_2$ in the original order $\preceq$. In this way, the new $\deg^+(w)$ for each $w \in V_C$ is not greater than the original $\deg^+(w)$. Therefore, the new $\deg^+(w)$ is smaller than or equal to $K+1$. Note that we do not need to update the $\deg^+$ for vertices in the original O_{K+1}. This explains why we insert V_C to the beginning of O_{K+1} rather than other positions.

The Algorithm: The pseudocode of the insertion algorithm is shown in Algorithm 2. While the algorithm is searching for V^*, it maintains the k-order at the same time. It consists of three phases: a) the preparing phase (lines 1-4); b) the core phase (lines 5-25); c) the ending phase (lines 26-33).

In the preparing phase, we set K as the smaller one of $\mathrm{core}(u)$ and $\mathrm{core}(v)$. By Theorem 3.2, only vertices in O_K may need to have core numbers updated. The new edge (u, v) then is inserted to G, and $\deg^+(u)$ is increased by 1 to reflect the insertion of (u, v), assuming $u \preceq v$.

In the core phase, if $\deg^+(u) \leq K$, O_K is still valid and no core number needs to be updated (Lemma 5.2). We can terminate the algorithm (line 5). On the other hand, if $\deg^+(u) > K$, we deal with the 3 cases using a while loop (lines 8-25), where each iteration contains three conditional branches, corresponding to **Case-1** (lines 10-13), **Case-2a** (lines 15-19) and **Case-2b** (lines 21-25) discussed above. For **Case-2b**, we call Algorithm 3 (line 24) to find all vertices in V_C that are not candidates any more.

In the ending phase, V_C contains all and only vertices in V^*. core thus is increased by 1 for each vertex in V_C. Following that, vertices in V_C are inserted to the beginning of O_{K+1} as discussed above. O'_K then becomes the O_K of G'. We update mcd for all relevant vertices in line 33 for edge removals in the future.

Correctness of the Algorithm: We show the correctness of Algorithm 2 in the following theorem. The proof can be obtained from the discussion above and is omitted here.

Theorem 5.1: *Algorithms 2 and 3 correctly update* core *and correctly maintain* O_K *and* O_{K+1}.

Complexity Analysis: To analyze the time complexity of Algorithm 2, we make the following assumptions for the data structures used and we will provide more details about the implementation later.

Algorithm 2: OrderInsert $(G, (u,v))$

input : $G = (V, E)$: the input graph; (u, v): the edge to insert

/* Preparing Phase */
1 $K \leftarrow \min\{\text{core}(u), \text{core}(v)\}$;
2 let the vertices in O_K be $v_1, v_2, \cdots, v_{|O_K|}$ in k-order;
3 $G' \leftarrow G \cup \{(u, v)\}$;
4 $\deg^+(u) \leftarrow \deg^+(u) + 1$ assuming $u \preceq v$;
/* Core Phase */
5 **if** $\deg^+(u) \leq K$ **then** **return**;
6 $O'_K \leftarrow \emptyset; V_C \leftarrow \emptyset$;
7 $i \leftarrow 1$;
8 **while** $i \leq |O_K|$ **do**
9   **if** $\deg^*(v_i) + \deg^+(v_i) > K$ **then**
10   remove v_i from O_K and append it to V_C;
11   **for each** $w \in \text{nbr}(v_i)$ **such that**
     $\text{core}(w) = K \wedge v_i \preceq w$ **do**
12        $\deg^*(w) \leftarrow \deg^*(w) + 1$;
13      $i \leftarrow i + 1$;
14   **else if** $\deg^*(v_i) = 0$ **then**
15   let $v_j \in O_K$ be the first vertex for $v_i \preceq v_j$ such that $\deg^*(v_j) > 0$ or $\deg^+(v_j) > K$;
16   **if** such v_j does not exist **then** $j \leftarrow |O_K| + 1$;
17   remove $v_i, \cdots, v_{j-1}$ from O_K;
18   append $v_i, \cdots, v_{j-1}$ to O'_K in order;
19      $i \leftarrow j$;
20   **else**
21   remove v_i from O_K and append to O'_K;
22      $\deg^+(v_i) \leftarrow \deg^+(v_i) + \deg^*(v_i)$;
23      $\deg^*(v_i) \leftarrow 0$;
24   RemoveCandidates (G', V_C, O'_K, v_i, K);
25      $i \leftarrow i + 1$;

/* Ending Phase */
26 $V^* \leftarrow V_C$;
27 **for each** $w \in V^*$ **do**
28    $\deg^*(w) \leftarrow 0$;
29    $\text{core}(w) \leftarrow \text{core}(w) + 1$;
30 insert vertices in V^* to the beginning of O_{K+1} in k-order;
31 let the k-order for O_K in G' be O'_K;
32 $G \leftarrow G'$;
33 update mcd accordingly;

Algorithm 3: RemoveCandidates(G', V_C, O'_K, w, K)

1 $\mathcal{Q} \leftarrow$ an empty queue;
2 **for each** $w' \in \text{nbr}(w)$ **such that** $w' \in V_C$ **do**
3    $\deg^+(w') \leftarrow \deg^+(w') - 1$;
4   **if** $\deg^+(w') + \deg^*(w') \leq K$ **then**
5      $\mathcal{Q}.\text{enqueue}(w')$;

6 **while** $\mathcal{Q} \neq \emptyset$ **do**
7    $w' \leftarrow \mathcal{Q}.\text{dequeue}()$;
8    $\deg^+(w') \leftarrow \deg^+(w') + \deg^*(w')$; $\deg^*(w') \leftarrow 0$;
9   remove w' from V_C;   append w' to O'_K;
10   **for each** $w'' \in \text{nbr}(w')$ **such that** $\text{core}(w'') = K$ **do**
11   **if** $w \preceq w''$ **then**
12        $\deg^*(w'') \leftarrow \deg^*(w'') - 1$;
13   **else if** $w' \preceq w'' \wedge w'' \in V_C$ **then**
14        $\deg^*(w'') \leftarrow \deg^*(w'') - 1$;
15   **if** $\deg^*(w'') + \deg^+(w'') \leq K \wedge w'' \notin \mathcal{Q}$ **then**
16          $\mathcal{Q}.\text{enqueue}(w'')$;
17   **else if** $w'' \in V_C$ **then**
18        $\deg^+(w'') \leftarrow \deg^+(w'') - 1$;
19   same codes as lines 15 to 16;

- For each k, we associate O_k with a data structure $\mathcal{A}_k$. For $\forall\, u, v \in O_k$, we can test whether $u \preceq v$ using $\mathcal{A}_k$ in $\mathcal{O}(\log |O_k|)$ time. Since we need to move vertices from O_K to O_{K+1}, $\mathcal{A}_k$ supports insertion (resp. removal) of a single vertex to (resp. from) $\mathcal{A}_k$ in $\mathcal{O}(\log |O_k|)$ time.
- A data structure $\mathcal{B}$ records all vertices $v_j \in O_K$ that $v_i \preceq v_j$ and $(\deg^*(v_j) > 0 \vee \deg^+(v_j) > K)$. $\mathcal{B}$ supports "jumping" by finding v_j (line 15 of Algorithm 2) in $\mathcal{O}(1)$ time. $\mathcal{B}$ supports insertion of a single vertex in $\mathcal{O}(\log |O_K|)$ time such that once $\deg^*(w)$ in line 12 becomes non-zero or $\deg^+(u)$ in line 4 becomes greater than K, we can insert w and u to $\mathcal{B}$ efficiently. $\mathcal{B}$ supports removal of a single vertex in $\mathcal{O}(\log |O_K|)$ time such that in line 12 of Algorithm 3, we can remove w'' from $\mathcal{B}$ efficiently once $\deg^*(w'')$ is decreased to 0, or remove v_i from $\mathcal{B}$ after v_i is processed in Algorithm 2.
- We assume moving vertices between O_K and O'_K or between O_K and O_{K+1} is done in $\mathcal{O}(1)$ time.

We denote the set of vertices in **Case-1** and **Case-2b** as V^+. In other words, V^+ consists of vertices that enter the first branch (line 9 in Algorithm 2) or the third branch (line 20). Intuitively, this is the set of vertices that we need to visit. The vertices in **Case-2a** are not considered because we do not need to expand them, i.e., we do not need to access their adjacent

edges. In addition, for **Case-2a**, the algorithm enters the second branch (line 14) at most $|V^+| + 1$ times, because each time the algorithm enters the second branch, it will either enter the first or the third branch or terminate in the next iteration. We emphasize that $V^* \subseteq V^+$. We have the following theorem for the complexity of the algorithm.

Theorem 5.2: *The time complexity of the insertion algorithm is* $\mathcal{O}(\sum_{v \in V^+} \deg(v) \cdot \log \max\{|O_K|, |O_{K+1}|\})$.

The size of V^+: We have the following two observations for V^+. First, if a vertex v enters the third branch, our algorithm in this case will decrease $\deg^+$ and $\deg^*$ only for vertices that have been in V^+. Therefore, this branch does not introduce new candidates to V^+. Second, if a vertex v enters the first branch, it will increase $\deg^*$ of its neighbors that are after it in the k-order. These neighbors may be visited and expanded (thus belong to V^+) later by the algorithm. Based on this observation, we introduce the notion *order core* below.

Definition 5.4: (order core) For each $u \in V(G)$, its order-core, denoted as $\text{oc}(u)$, is recursively defined as follows.

$$\text{oc}(u) = \{u\} \cup \left(\bigcup\nolimits_{w \in \text{nbr}(u) \,\wedge\, \text{core}(u) = \text{core}(w) \,\wedge u \preceq w} \text{oc}(w)\right)$$

Essentially, for each $w \in \text{oc}(u)$, there is a path $w_0, w_1, \cdots, w_t$ such that $w_0 = u$, $w_t = w$, $(w_i, w_{i+1}) \in E$ and $w_i \preceq w_{i+1}$ for $0 \leq i < t$. The lemma below relates V^+, $\text{oc}(u)$ together.

Lemma 5.4: *Suppose* (u, v) *is the newly inserted edge. (1) If* $\text{core}(u) < \text{core}(v)$, $V^+ \subseteq \text{oc}(u)$. *(2) If* $\text{core}(u) = \text{core}(v)$, $V^+ \subseteq \text{oc}(u) \cup \text{oc}(v)$.

By Lemma 5.4, the size of order cores upper bounds the maximum possible size of V^+. Recall that oc is the max # of vertices that our insertion algorithm visits and expands, pc (Definition 4.1) is the max # of vertices that the traversal insertion algorithm visits and expands, and sc (Section III) is the max # of vertices that need to visit and expand by Theorem 3.2. We show the cumulative distribution of oc for Patents and Orkut in Fig. 5. From Fig. 5, oc has much smaller variation in the size than sc and pc. While 90% of vertices

Algorithm 4: OrderRemoval $(G, (u, v))$

input : $G = (V, E)$: the input graph; (u, v): the edge to
remove
1   $K \leftarrow \min\{\mathsf{core}(u), \mathsf{core}(v)\}$;
2   $G \leftarrow G \setminus (u, v)$;
3   **if** $\mathsf{core}(u) \leq \mathsf{core}(v)$ **then** $\mathsf{mcd}(u) \leftarrow \mathsf{mcd}(u) - 1$;
4   **if** $\mathsf{core}(u) \geq \mathsf{core}(v)$ **then** $\mathsf{mcd}(v) \leftarrow \mathsf{mcd}(v) - 1$;
5   find V^* using the routine used in the traversal removal
    algorithm (described in Section IV-B) and assume core has
    been correctly updated;
    /* Update the k-order below   */
6   **for each** $w \in V^*$ in the order they are inserted to V^* **do**
7      $\deg^+(w) \leftarrow 0$;
8   **for each** $w' \in \mathsf{nbr}(w)$ **do**
9   **if** $\mathsf{core}(w') = K \wedge w' \preceq w$ **then**
10          $\deg^+(w') \leftarrow \deg^+(w') - 1$;
11   **if** $\mathsf{core}(w') \geq K \vee w' \in V^*$ **then**
12          $\deg^+(w) \leftarrow \deg^+(w) + 1$;
13      $V^* \leftarrow V^* \setminus \{w\}$;
14   remove w from O_K and append it to O_{K-1};
15   update mcd accordingly;

---

have their oc in the order of hundreds or less, 90% vertices
have their pc and sc in the order of 10,000 or less. In other
words, our algorithm visits much less vertices than the traversal
insertion algorithm does in the worst case.

In order to investigate the practical size of V^+, for
the 11 real graphs tested (Table I), we calculate the ratio
$\frac{\text{sum of } |V^+| \text{ over all insertions}}{\text{sum of } |V^*| \text{ over all insertions}}$ for our algorithm after inserting
100,000 edges one by one to each of the graphs. The results are
shown in Fig. 2. As can be seen, for all graphs, the ratios are
smaller than 4, which indicates the efficiency of our algorithm.
In addition, we show the distribution of $|V^+|$ in Fig. 1. The
distributions show that the proportion that $|V^+| > 100$ is
negligible for all the 11 graphs, which indicates that our
algorithm is efficient and of small performance variation.

C. The Order-Removal Algorithm

The removal algorithm is presented in Algorithm 4. First,
we remove (u, v) from G and update mcd to reflect the
removal of (u, v). The algorithm adopts the same routine used
in the traversal removal algorithm (Section IV-B) to find V^*.
We maintain the k-order as follows. For each $w \in V^*$ in the
order they are inserted to V^*, we update the $\deg^+$ of w and
its neighbors, remove w from O_K, and insert w to the end
of O_{K-1}. This is different from the insertion case, where we
insert vertices to the beginning of O_{K+1}. Finally, we update
mcd (line-15) to reflect the change of core.

Algorithm 4 correctly updates core because we adopt the
same routine used in the traversal removal algorithm to find
V^*. For O_K and O_{K-1}, we have the following theorem.

Theorem 5.3: *Algorithm 4 correctly maintains O_K and O_{K-1}.*

We adopt the similar idea used in the traversal removal
algorithm, since it requires only $\mathcal{O}(\sum_{v \in V^*} \deg(v))$ to find V^*,
as discussed in Section IV-B. On the other hand, the critical
difference between our OrderRemoval (Algorithm 4) and the
traversal removal algorithm is the index under maintenance. In
the traversal removal algorithm, it needs to maintain the pcd
values of the vertices, while in our algorithm, we instead main-
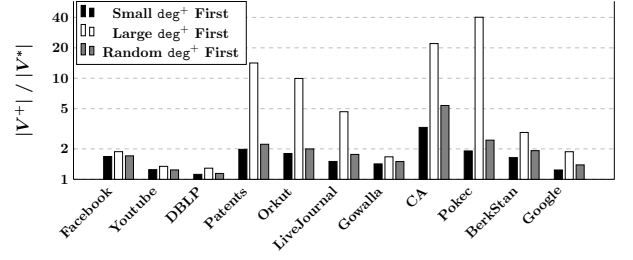tain the k-order (O_K and O_{K-1}). As shown in Section IV-B,

Fig. 9: Comparison of Heuristics to Generate the k-Order

the cost of maintaining pcd is usually large and hurts the total
performance significantly. Different from pcd, k-orders can be
maintained much more efficiently. Specifically, maintaining the
k-order requires only $\mathcal{O}(\log |O_K| \cdot \sum_{w \in V^*} \deg_K(w) + |V^*| \cdot \log |O_{K-1}|)$ time in the worst case, where $\deg_K(w)$ counts
the number of neighbors w' of w that $\mathsf{core}(w') = K$ and is
usually small. Formally, we have the following theorem for
the complexity of the removal algorithm.

Theorem 5.4: *Algorithm 4 runs in* $\mathcal{O}(\sum_{w \in V^*} \deg(w) + \log |O_K| \cdot \sum_{w \in V^*} \deg_K(w) + |V^*| \cdot \log |O_{K-1}|)$ *time.*

VI. IMPLEMENTATION

Generation of the k-Order: We generate the k-order for a
graph G based on CoreDecomp (Algorithm 1) by inserting the
following code immediately after line 7 of Algorithm 1.

---

append u to O_{k-1}; $\deg^+(u) \leftarrow \deg(u)$;

In addition, we use a "small $\deg^+$ first" heuristic during the
generation process of O_ks, i.e., we always choose a vertex
that has the minimum $\deg^+$ to append to O_{k-1}. If there is a
tie, we break the tie arbitrarily. The justification is as follows:
In Algorithm 2, to avoid a large $|V^+|$, the only way is to
reduce the probability of entering the first conditional branch
(line 9). By putting vertices with smaller $\deg^+$ before other
vertices, when the algorithm visits the vertices in order, it
will first encounter those vertices with smaller $\deg^+$. These
vertices intuitively would be less likely to enter the first branch,
thus do not introduce new candidates to V^+, which in return
reduces the number of vertices to visit. We emphasize that
the algorithm with "small $\deg^+$ first" heuristic can still run in
linear time by adjusting the implementation by [4]. To verify
the effectiveness of our heuristic, we compare it with another
two heuristics, namely, "large $\deg^+$ first" and "random $\deg^+$
first". By "large $\deg^+$ first", we append to O_{k-1} a vertex with
largest $\deg^+$, while by "random $\deg^+$ first", we append to
O_{k-1} a vertex randomly as long as its $\deg^+$ is smaller than k.
We show $\frac{\text{sum of } |V^+| \text{ over all insertions}}{\text{sum of } |V^*| \text{ over all insertions}}$ for three heuristics in
Fig. 9 after inserting 100,000 edges. From the figure, we can
see that the "small $\deg^+$ first" heuristic consistently performs
better in all 11 graphs tested.

Implementation: In order to traverse O_k efficiently, O_k is
implemented as a doubly linked list. Recall that we associate
O_k with a data structure $\mathcal{A}_k$ to test $u \preceq v$ when both are in
O_k. In addition, we use a data structure $\mathcal{B}$ such that jumping to
v_j (line 15 of Algorithm 2) takes $\mathcal{O}(1)$ time. In order to make

both OrderInsert (Algorithm 2) and OrderRemoval (Algorithm 4) efficient, there are two main issues: (A) how to implement $\mathcal{A}_k$, and (B) how to implement $\mathcal{B}$.

(A) How to implement $\mathcal{A}_k$: Here, the data structure is needed to test $u \preceq v$ when both are in the same O_k; that is, $\mathrm{core}(u) = \mathrm{core}(v)$. On selecting an appropriate data structure for such a purpose, we take the following observation into consideration.

Observation 6.1: *In algorithm* OrderInsert, *it is possible that $u \preceq v$ in the k-order $\preceq$ of G is changed to $v \preceq' u$ in the new order $\preceq'$ after G is updated by inserting/removing an edge.*

We explain how it happens. Consider processing vertices in O_k from left to right one-by-one as an example. Suppose $u \preceq v$. Initially, suppose u is in **Case-1**. As a result, by the OrderInsert algorithm, u is appended to V_C instead of O'_K as a candidate to be in V^*. While u is in V_C, the OrderInsert algorithm processes v, finds that v is in **Case-2**, and thus appends v to O'_K. Some time later, it is possible that u is found not to be a candidate any more when the algorithm processes a vertex w, which is in **Case-2b**. Note that w may be v. Then u is removed from V_C and also appended to O'_K. This causes $v \preceq' u$ in the new k-order $\preceq'$. We emphasize that there are at most $|V^+|$ such us on a single edge insertion because V^+ includes all vertices that are in **Case-1** in the OrderInsert algorithm. We discuss how to support $\mathcal{A}_k$ below.

In current implementation, we encode $\mathcal{A}_k$ using an *order statistics tree*, where each tree node holds exactly one vertex. The following invariant is maintained: for any two vertices u and v in O_k, if $u \preceq v$, then (1) u is in the left subtree of v, or (2) v is in the right subtree of u, or (3) there exists $w \in O_k$ such that u is in the left subtree of w and v is in the right subtree of w. By associating each node with the size of the subtree rooted at the node, we are able to find the rank of a vertex in O_k in $\mathcal{O}(\log |O_k|)$ time [7]. Such a method faces an issue. Given an order statistics tree, in order to find the rank of a vertex, we need to first locate the node containing the vertex, starting from the root. But we cannot decide which child pointer to follow to find the node, since we do not have the rank information yet. In other words, given a vertex, in order to get the rank of the vertex using the tree, we should first know the rank of the vertex. We propose a mechanism to address this issue by additionally creating a one-to-one mapping between vertices in O_k and nodes in the order statistics tree. In this way, locating the target node in the order statistics tree can be done easily. It is worth noting that in our algorithm, vertices are inserted either to the beginning of O_{K+1} for insertion or the end of O_{K-1} for removal. Accordingly, we only need to follow either the left child pointers or the right child pointers to insert the vertex to the order statistics tree. Therefore, the issue mentioned above is avoided. The post-processing to make the tree balanced is similar to that of an ordinary tree. Both insertion and removal of a vertex takes $\mathcal{O}(\log |O_k|)$ time.

We show below that $\mathcal{A}_k$s deal with Observation 6.1 well. To make it consistent with the new k-order O'_k, we adjust the position of u (refer to Observation 6.1) in the order statistics tree as follows. First, we remove u from the tree. Suppose u' is the previous vertex of u in O'_k. We insert u to the right subtree of u' by following only left child pointers such that u is the successor of u' in the tree. Local rotations are then taken

to make the tree valid. The total cost for adjusting positions thus is $\mathcal{O}(|V^+|\log|O_k|)$ because there are at most $|V^+|$ such us and each takes $\mathcal{O}(\log|O_k|)$ time. This additional cost does not affect the complexity of Theorem 5.2.

Theorem 6.1: *The total space cost of $\mathcal{A}_k$s is $\mathcal{O}(n)$ and for each k, $\mathcal{A}_k$ can be created in $\mathcal{O}(|O_k|\log|O_k|)$ time by inserting vertices in O_k one by one.*

(B) How to implement $\mathcal{B}$: $\mathcal{B}$ is implemented as a min-heap. $\mathcal{B}$ maintains a set of $(\mathrm{rank}(w), w)$ pairs with $\deg^*(w) \neq 0$ or $\deg^+(w) > K$ (**Case-1** and **Case-2b**), and uses the rank as the key. When we need to find the v_j (line 15 of Algorithm 2), the top pair of $\mathcal{B}$ is returned in $\mathcal{O}(1)$ time. It requires $\mathcal{O}(\log|\mathcal{B}|) = \mathcal{O}(\log|O_K|)$ time to insert a pair to $\mathcal{B}$ since only vertices in O_K will be inserted to $\mathcal{B}$. It requires $\mathcal{O}(\log|\mathcal{B}|)$ time to remove arbitrary pair from $\mathcal{B}$.

VII. PERFORMANCE STUDIES

We have conducted experimental studies using 11 real large graphs, and we report the performance of our algorithms by comparing with the traversal algorithms. All algorithms are implemented in C++ and compiled by g++ compiler at -O2 optimization level. For the new order-based algorithms, we implement the order statistics tree on top of *treaps*. For the traversal algorithms, we apply the enhancement proposed in [14], which is the journal version of [13]. The enhancement consists of exploiting neighborhood of higher hops to improve the pruning power of pcd, which considers only 2-hop neighborhood of a vertex, at a higher maintenance cost. Note that the traversal removal algorithm requires only mcd and thus its performance degrades for higher hops. We set the hop count h in the range $\{2, 3, 4, 5, 6\}$ as in [14] and denote each version as Trav-h. All experiments are conducted on a Linux machine with Intel i7-4790 CPU and 32 GB main memory.

Datasets: We use 11 datasets publicly accessible, which are shown in Table I. These real datasets cover a wide range of graphs used in different applications with different properties. Facebook, Youtube, and DBLP are three temporal graphs, whose edges are labeled with time stamps to show when they are inserted, and can be downloaded from Konect (http://konect.uni-koblenz.de). The remaining eight graphs, which can be downloaded from SNAP (https://snap.stanford.edu), include Social Networks (LiveJournal, Pokec, and Orkut), Citation Network (Patents), Web Graphs (BerkStan and Google), Road Network (CA), and Location-Based Social Network (Gowalla). Directed graphs are converted to undirected ones in our testing. The statistics are shown in Table I. The cumulative distribution of core numbers for each graph is shown in Fig. 10a.

A. Order-Based vs Traversal

We compare the performance of our order-based algorithms with the traversal algorithms, which are the state-of-the-art. Here, we select 100,000 edges out of each of the 11 graphs as follows. For the three graphs with time stamps on edges, Facebook, Youtube and DBLP, because each edge is associated with a time stamp, we select the latest 100,000 edges, i.e., the edges with maximum time stamps. For each of the remaining eight graphs, we randomly sample 100,000

Dataset	$n = |V|$	$m = |E|$	avg. deg	max k
Facebook	63,731	817,035	25.64	52
Youtube	3,223,589	9,375,374	5.82	88
DBLP	1,314,050	5,362,414	8.16	118
Patents	3,774,768	16,518,947	8.75	64
Orkut	3,072,441	117,185,083	76.28	253
LiveJournal	4,846,609	42,851,237	17.68	372
Gowalla	196,591	950,327	9.67	51
CA	1,965,206	2,766,607	2.82	3
Pokec	1,632,803	22,301,964	27.32	47
BerkStan	685,230	6,649,470	19.41	201
Google	875,713	4,322,051	9.87	44

TABLE I: Datasets

(a) The cumulative distribution of core numbers

(b) Cumulative distribution of K of the 100,000 edges sampled
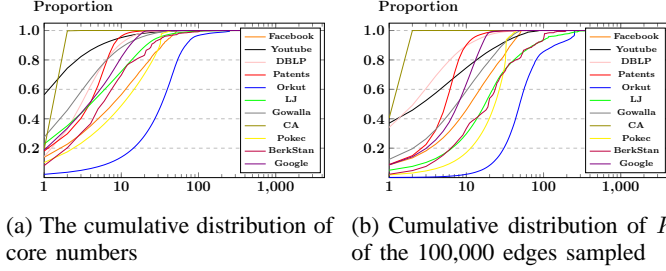
Fig. 10: Statistics of the Graphs and Edges Tested

edges. For each graph, we show the cumulative distribution of K of the 100,000 edges sampled in Fig. 10b, where $K = \min\{\text{core}(u), \text{core}(v)\}$ given an edge (u, v). Fig. 10b shows that the edges to test cover k-cores in wide range, implying they are suitable for testing. For each graph, we measure the accumulated time for inserting these 100,000 edges one by one, and then measure the accumulated time for removing these edges from the graph. The results are shown in TABLE II.

Edge Insertion: We compare OrderInsert (Algorithm 2) with the traversal insertion algorithm [13] ([14]). The results are shown in TABLE II. OrderInsert significantly outperforms the traversal insertion algorithm in all the datasets tested. For Pokec, the speedup even achieves up to 2,083 times. For Patents and CA, we further test with higher h for the traversal insertion algorithm to get the optimal efficiency and the results are 810.00s ($h = 6$) for Patents and 1.08s ($h = 7$) for CA, which are still slower than OrderInsert. As shown in Fig. 2, OrderInsert visits much less vertices than Trav-2. This confirms the efficiency of OrderInsert algorithm. In addition, we emphasize that OrderInsert has much smaller performance variation between edges inserted, as shown in Fig. 1.

Edge Removal: We compare OrderRemoval (Algorithm 4) with the traversal removal algorithm [13] ([14]). The results are shown in TABLE II. We observe that Trav-2 is more efficient than those with higher h, and OrderRemoval outperforms Trav-2 in all datasets except CA. Here, the average degree of CA is only 2.82 (≈ 3), as shown in TABLE I. Such a small average degree indicates that the cost to maintain pcd is low, because the number of vertices whose pcd need to be updated after an edge update is small. On the other hand, OrderRemoval outperforms Trav-2 in all other graphs tested, especially Pokec and Orkut, because Pokec and Orkut have the highest average degrees among all the datasets.
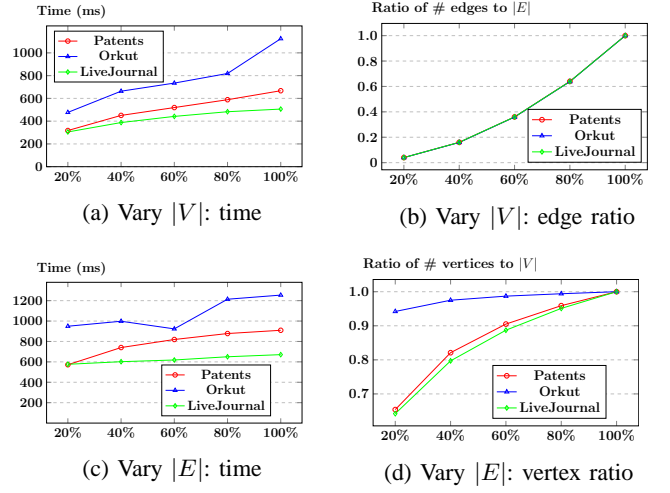

(a) Vary $|V|$: time

(b) Vary $|V|$: edge ratio

(c) Vary $|E|$: time

(d) Vary $|E|$: vertex ratio

Fig. 11: The Scalability of the OrderInsert Algorithm

Index Space and Index Creation: Our order-based algorithm requires $\mathcal{O}(n)$ space, as shown in Theorem 6.1. In our current implementation, the order-based algorithm needs about five times of the space used by Trav-2 and needs only about double time to create the index (including computing the initial core numbers). When the hop count h becomes larger, the space and time required by the traversal algorithms also become higher. See [18]. We emphasize that index creation is a one-time cost. Moreover, in view of the insertion/removal performance improvement brought, such a tradeoff is appropriate.

B. Scalability Testing

We test the scalability of OrderInsert using the largest three datasets, i.e., Patents, Orkut, and LiveJournal. We vary the number of vertices $|V|$ and the number of edges $|E|$ respectively by randomly sampling vertices and edges at rates from 20% to 100%. When sampling vertices, we use the subgraph induced by the vertices. When sampling edges, we keep the incident vertices of the edges. From each subgraph sampled, we further sample 100,000 edges for testing. We show the results in Fig. 11. Fig. 11a shows the total time (ms) to insert the sampled 100,000 edges one by one and update core numbers, and Fig. 11b shows the ratio of the number of edges in each sampled subgraph to the number of edges in the original graph, while varying $|V|$. As can be seen, the time taken by OrderInsert grows smoothly while the number of edges increase rapidly. Fig. 11c shows the total time (ms) to insert the sampled 100,000 edges and update core numbers, and Fig. 11d shows the ratio of the number of vertices in each sampled subgraph to the number of vertices in the original graph, while varying $|E|$. Similarly, OrderInsert performs well while the number of vertices increase rapidly. We do not show the scalability results for OrderRemoval because it only relies on V^* and their neighbors, as shown in Theorem 5.4. The scalability thus is not an issue of OrderRemoval.

C. Stability Testing

We update the k-order when an edge inserted. An issue is whether the effectiveness of the k-order will be stable after a large number of edge insertions. We test the stability of the algorithm using the largest three datasets, i.e., Patents, Orkut, and LiveJournal. The stability testing is conducted as

	Insert						Remove					
Dataset	OrderInsert	Trav-2	Trav-3	Trav-4	Trav-5	Trav-6	OrderRemoval	Trav-2	Trav-3	Trav-4	Trav-5	Trav-6
Facebook	**0.16**	*3.52*	4.07	5.91	10.52	16.95	**0.10**	*0.50*	1.63	4.14	9.70	17.77
Youtube	**0.26**	*2.51*	2.88	4.01	6.13	9.71	**0.28**	*0.61*	1.42	3.19	6.28	11.32
DBLP	**0.16**	1.80	*1.20*	2.31	6.32	17.65	**0.11**	*0.21*	0.61	1.88	5.49	15.78
Patents	**0.88**	2,944.14	1,805.98	1,173.20	845.93	*810.00*	**0.38**	*0.92*	4.22	18.57	75.06	276.37
Orkut	**1.14**	954.36	793.82	*780.69*	996.43	1,576.63	**0.71**	*7.75*	36.80	136.78	428.85	1,089.38
LiveJournal	**0.53**	149.56	90.93	*76.57*	125.29	285.50	**0.33**	*1.66*	6.59	24.56	86.10	233.92
Gowalla	**0.18**	*1.04*	1.37	2.21	3.78	6.38	**0.14**	*0.35*	0.84	1.82	3.45	6.22
CA	0.52	15.14	4.20	2.08	1.37	*1.11*	0.16	**0.08**	*0.13*	0.19	0.26	0.33
Pokec	**0.77**	1,726.04	*1,603.80*	1,650.37	1,876.48	2,338.78	**0.32**	*4.86*	53.13	259.93	756.40	1,652.88
BerkStan	**0.37**	*6.37*	7.29	9.37	13.14	16.19	**0.52**	*2.55*	5.04	8.33	12.45	17.34
Google	**0.37**	*1.01*	1.25	2.44	4.81	9.27	**0.25**	*0.46*	0.96	2.08	4.32	8.75

TABLE II: Performance Comparison (in seconds): Order-Based vs Traversal (The winner is in **bold**; the runner-up is in *italics*)
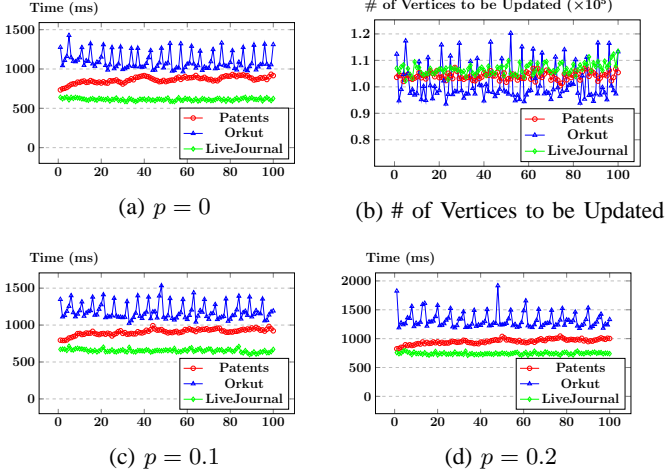

Fig. 12: The Stability of the OrderInsert Algorithm

follows. First, we randomly sample 10,000,000 edges from the graph and randomly partition them into 100 groups, where each group has 100,000 edges. Second, we reinsert these edges to the graph group by group. For each group, we measure the accumulated time used by OrderInsert to insert the edges one by one. Fig. 12a shows the results. The performance of the OrderInsert insertion algorithm is well bounded, for all three datasets. For Orkut graph, we observe that its result fluctuates frequently. This is due to the fact that the number of vertices whose core numbers need to be updated inside each group varies a lot, as can be seen in Fig. 12b. To make the experiment more practical, we conducted two additional experiments as follows. For each group, after an edge is inserted, with probability p we randomly remove an edge from the graph. In the experiments, we set p as 0.1 and 0.2 respectively and the total time measured also includes the time of removing edges. The results are similar to Fig. 12a, as shown in Fig. 12c and 12d.

VIII. CONCLUSIONS

In this paper, we study core maintenance to reduce the computational cost to compute k-cores for a graph when edges are inserted/removed dynamically. To further improve the efficiency of the state-of-the-art traversal algorithm, we propose a new order-based algorithm. The key is to explicitly maintain the k-order for a graph. With the k-order, we can identify the set of vertices that need to be updated efficiently, and we can maintain the k-order for a graph with small overhead when edges are inserted/removed. We confirm our approach by conducting extensive performance studies using 11 real large graphs. Our OrderInsert algorithm outperforms the traversal insertion algorithm up to 3 orders of magnitude.

REFERENCES

[1] H. Aksu, M. Canim, Y.-C. Chang, I. Korpeoglu, and O. Ulusoy. Distributed-core view materialization and maintenance for large dynamic graphs. *IEEE TKDE*, 26(10), 2014.

[2] J. I. Alvarez-Hamelin, L. DallAsta, A. Barrat, and A. Vespignani. How the k-core decomposition helps in understanding the internet topology. In *ISMA Workshop on the Internet Topology*, volume 1, 2006.

[3] J. I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani. Large scale networks fingerprinting and visualization using the k-core decomposition. In *Proc. of NIPS'06*, 2006.

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

[5] J. Cheng, Y. Ke, S. Chu, and M. T. Özsu. Efficient core decomposition in massive networks. In *Proc. of ICDE'11*, 2011.

[6] J. Cohen. Trusses: Cohesive subgraphs for social network analysis. *National Security Agency Technical Report*, page 16, 2008.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2009.

[8] A. Epasto, S. Lattanzi, and M. Sozio. Efficient densest subgraph computation in evolving graphs. In *Proc. of WWW'15*, 2015.

[9] W. Fan, J. Li, J. Luo, Z. Tan, X. Wang, and Y. Wu. Incremental graph pattern matching. In *Proc. of SIGMOD'11*, 2011.

[10] W. Khaouid, M. Barsky, V. Srinivasan, and A. Thomo. K-core decomposition of large networks on a single pc. *PVLDB*, 9(1), 2016.

[11] R.-H. Li, J. X. Yu, and R. Mao. Efficient core maintenance in large dynamic graphs. *IEEE TKDE*, 26(10), 2014.

[12] A. Montresor, F. De Pellegrini, and D. Miorandi. Distributed k-core decomposition. *IEEE TPDS*, 24(2), 2013.

[13] A. E. Sariyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and Ü. V. Çatalyürek. Streaming algorithms for k-core decomposition. *PVLDB*, 6(6), 2013.

[14] A. E. Sariyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and Ü. V. Çatalyürek. Incremental k-core decomposition: algorithms and evaluation. *VLDBJ*, 25(3), 2016.

[15] J. Wang and J. Cheng. Truss decomposition in massive networks. *PVLDB*, 5(9), 2012.

[16] D. Wen, L. Qin, Y. Zhang, X. Lin, and J. X. Yu. I/o efficient core graph decomposition at web scale. In *Proc. of ICDE'16*, 2016.

[17] H. Zhang, H. Zhao, W. Cai, J. Liu, and W. Zhou. Using the k-core decomposition to analyze the static structure of large-scale software systems. *The Journal of Supercomputing*, 53(2), 2010.

[18] Y. Zhang, J. X. Yu, Y. Zhang, and L. Qin. A fast order-based approach for core maintenance. *arXiv preprint arXiv:1606.00200*, 2016.

[19] A. D. Zhu, W. Lin, S. Wang, and X. Xiao. Reachability queries on large dynamic graphs: A total order approach. In *Proc. of SIGMOD'14*, 2014.