I/O Efficient Core Graph Decomposition: Application to Degeneracy Ordering

Dong Wen, Lu Qin, Ying Zhang, Xuemin Lin, *Fellow, IEEE*, and Jeffrey Xu Yu

Abstract—Core decomposition is a fundamental graph problem with a large number of applications. Most existing approaches for core decomposition assume that the graph is kept in memory of a machine. Nevertheless, many real-world graphs are too big to reside in memory. In this paper, we study I/O efficient core decomposition following a semi-external model, which only allows node information to be loaded in memory. We propose a semi-external algorithm and an optimized algorithm for I/O efficient core decomposition. To handle dynamic graph updates, we firstly show that our algorithm can be naturally extended to handle edge deletion. Then we propose an I/O efficient core maintenance algorithm to handle edge insertion, and an improved algorithm to further reduce I/O and CPU cost. In addition, based on our core decomposition algorithms, we further propose an I/O efficient semi-external algorithm for degeneracy ordering, which is an important graph problem that is highly related to core decomposition. We also consider how to maintain the degeneracy order. We conduct extensive experiments on 12 real large graphs. Our optimal core decomposition algorithm significantly outperforms the existing I/O efficient algorithm in terms of both processing time and memory consumption. They are very scalable to handle web-scale graphs. As an example, we are the first to handle a web graph with 978.5 million nodes and 42.6 billion edges using less than 4.2 GB memory. We also show that our proposed algorithms for degeneracy order computation and maintenance can handle big graphs efficiently with small memory overhead.

Index Terms—Core Decomposition, Degeneracy Order, Dynamic Graphs.

1 Introduction

Graphs have been widely used to represent the relationships of entities in a large spectrum of applications such as social networks, web search, collaboration networks, and biology. With the proliferation of graph applications, research efforts have been devoted to many fundamental problems in managing and analyzing graph data. Among them, the problem of computing the k-core of a graph has been recently studied [1–4]. Here, given a graph G, the k-core of G is the largest subgraph of G such that all the nodes in the subgraph have a degree of at least k [5]. For each node v in G, the core number of v denotes the largest k such that v is contained in a k-core. The core decomposition problem computes the core numbers for all nodes in G. Given the core decomposition of a graph G, the k-core of G for all possible k values can be easily obtained. There is a linear time in-memory algorithm, devised by Batagelj and Zaversnik [6], to compute core numbers of all nodes.

Applications. Core decomposition is widely adopted in many real-world applications, such as community detection [4, 7], network clustering [8], network topology analysis [5, 9], network visualization [10, 11], protein-protein network analysis [12, 13], and system structure analysis [14]. In addition, many researches are devoted to the core decomposition for specific kinds of networks [15–20]. Moreover, due to the elegant structural property of a k-core and the linear solution for core decomposition, a large number of graph problems use core decomposition as a subroutine or a preprocessing step, such as clique finding [21], dense subgraph discovery [22, 23], approximation of betweeness scores [24], and some variants of community search problems [25, 26].

Motivation. Despite the large amount of applications for core decomposition in various networks, most of the solutions for core decomposition assume that the graph is resident in the main memory of a machine. Nevertheless, many real-world graphs are big and may not reside entirely in the main memory. For example, the Facebook social network contains 1.32 billion nodes and 140 billion edges (http://newsroom.fb.com/company-info) and a subdomain of the web graph Clueweb contains 978.5 million nodes and 42.6 billion edges (http://law.di.unimi.it/datasets.php). In the literature, the only solution to study I/O efficient core decomposition is EMCore proposed by Cheng et al. [1], which allows the graph to be partially loaded in the main memory. EMCore adopts a graph partition based approach and partitions are loaded into main memory whenever necessary. However, EMCore cannot bound the size of the memory and to process many real-world graphs, EMCore still loads most edges of the graph in the main memory. This makes EMCore unscalable to handle web-scale graphs. In addition, many real-world graphs are usually dynamically updating. The complex structure used in EMCore makes it very difficult to handle graph updates incrementally.

Our Solution. In this paper, we address the drawbacks of the existing solutions for core decomposition and propose new algorithms with a guaranteed memory bound. Specifically, we adopt a semi-external model. It assumes that the nodes of the graph, each of which is associated with a small constant amount of information, can be loaded in main memory while the edges are stored on disk. We find that this assumption is practical in a large number of real-world web-scale graphs, and widely adopted to handle other graph problems [27–29]. Based on such an assumption, we are able to handle core decomposition I/O efficiently using very simple

- *Dong Wen is with the University of Technology Sydney, Australia (email: dong.wen@student.uts.edu.au).*
- *Lu Qin is with the University of Technology Sydney, Australia (email: lu.qin@uts.edu.au).*
- *Ying Zhang is with the University of Technology Sydney, Australia (email: ying.zhang@uts.edu.au).*
- *Xuemin Lin is with the East China Normal University, China, and University of New South Wales, Australia (email: lxue@cse.unsw.edu.au).*
- *Jeffrey Xu Yu is with the Chinese University of Hong Kong, China (email: yu@se.cuhk.edu.hk).*

structures and data access mechanisms. The I/O complexity of our algorithm is $O(\frac{l \cdot (m+n)}{B})$ where l is the iteration number, m and n are the numbers of nodes and edges of the graph and B is the disk block size. We refer to our proposed approace as "I/O efficient" because (1) l is quite small in practice and (2) in most of the iterations, we only need to access a small part of the graph on disk. For example, to process a dataset with 978.5 million nodes and 42.6 billion edges used in our experiments, the total number of I/Os used by our algorithm is only 3 times the value of $\frac{m+n}{B}$. The semi-external model can also be used to handle graph updates.

Degeneracy Ordering. Degeneracy ordering, which is highly related to core decomposition, is a basic graph problem in graph theory [30]. The degeneracy order of a graph G is a total order of nodes in the graph such that every node has limited number of neighbors (denoted as $d(G)$) in its right-side nodes in the order. Degeneracy ordering is applied in many applications such as graph coloring [31] and graph clustering [32, 33]. In graph coloring, if we follow the degeneracy order to color nodes in the graph, we can guarantee that the number of colors used is bounded by $d(G)+1$ [31]. In graph clustering, a typical problem is to find a subgraph with maximum density. If we construct subgraphs following the degeneracy order, we can find a 2-approximation solution to this problem [33]. Some other applications can be found in [30].

In the literature, all algorithms for degeneracy ordering assume that the graph can reside in memory. In this paper, we find that, under the semi-external model, the degeneracy order can be computed based on our I/O efficient algorithm for core decomposition, and we can also maintain the degeneracy order incrementally on dynamic graphs based on our I/O efficient core maintenance algorithm.

Contributions. In the following, we summarize the main contributions of this paper. The preliminary version of this work is published in [34].

(1) The first I/O efficient core decomposition algorithm with a memory guarantee. We propose an I/O efficient core decomposition algorithm following the semi-external model. Our algorithm only keeps the core numbers of nodes in memory and updates the core numbers iteratively until convergence. In each iteration, we only require sequential scans of edges on disk. To the best of our knowledge, this is the first work for I/O efficient core decomposition with a memory guarantee.

(2) Several optimization strategies to largely reduce the I/O and CPU cost. Through further analysis, we observe that when the number of iterations increases, only a very small proportion of nodes have their core numbers updated in each iteration, and thus scanning all edges on disk in each iteration will result in a large number of waste I/O and CPU time. Therefore, we propose optimization strategies to reduce this cost. Our first strategy is based on the observation that the update of the core number of a node should be triggered by the update of the core number of at least one of its neighbors in the graph. Our second strategy further maintains more node information. As a result, we can completely avoid waste I/Os and core number computations, in the sense that each I/O is used in a core number computation that is guaranteed to update the core number of the corresponding node.

(3) The first I/O efficient core decomposition algorithm to handle graph updates. We consider dynamic graphs with edge deletion and insertion. Our semi-external algorithm can naturally support edge deletion with a simple algorithm modification. For edge insertion, we first take advantage of some graph properties already used in existing in-memory algorithms [2, 3] to handle graph updates for core decomposition. We propose a two-phase semi-external algorithm to handle edge insertion using these graph properties. We further explore some new graph properties, and propose a new one-phase semi-external algorithm to largely reduce the I/O and CPU time for edge insertion. To the best of our knowledge, this is the first work for I/O efficient core maintenance on dynamic graphs.

(4) The first I/O efficient algorithms for degeneracy order computation and maintenance. We propose I/O efficient algorithms for degeneracy order computation and degeneracy order maintenance respectively under the semi-external model. We significantly improve their efficiency by making use of our core decomposition algorithm and core maintenance algorithm respectively. To the best of our knowledge, this is the first work for I/O efficient degeneracy order computation and maintenance.

(5) Extensive performance studies. We conduct extensive performance studies using 12 real graphs with various graph properties to demonstrate the efficiency of our algorithms. We compare our algorithm, for memory-resident graphs, with EMCore [1] and the in-memory algorithm [6]. Both our core decomposition and core maintenance algorithms are much faster and use much less memory than EMCore. In many datasets, our algorithms for core decomposition and maintenance are even faster than the in-memory algorithm due to the simple structure and data access model used. Our algorithms for degeneracy order computation and maintenance are also efficient and scalable to handle big graphs with bounded memory under the semi-external model.

2 Problem Statement

Consider an undirected and unweighted graph $G = (V, E)$, where $V(G)$ represents the set of nodes and $E(G)$ represents the set of edges in G. We denote the number of nodes and the number of edges of G by n and m respectively. We use $\mathsf{nbr}(u, G)$ to denote the set of neighbors of u in G, i.e., $\mathsf{nbr}(u, G) = \{v|(u, v) \in E(G)\}$. The degree of a node $u \in V(G)$, denoted by $\mathsf{deg}(u, G)$, is the number of neighbors of u in G, i.e., $\mathsf{deg}(u, G) = |\mathsf{nbr}(u, G)|$. For simplicity, we use $\mathsf{nbr}(u)$ and $\mathsf{deg}(u)$ to denote $\mathsf{nbr}(u, G)$ and $\mathsf{deg}(u, G)$ respectively. A graph G' is a subgraph of G, denoted by $G' \subseteq G$, if $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$. Given a set of nodes $V_c \subseteq V$, the induced subgraph of V_c, denoted by $G(V_c)$, is a subgraph of G such that $G(V_c) = (V_c, \{(u, v) \in E(G)|u, v \in V_c\})$.

Definition 2.1: (k-Core) Given a graph G and an integer k, the k-core of graph G, denoted by G_k, is a maximal subgraph of G in which every node has a degree of at least k, i.e., $\forall v \in V(G_k), d(v, G_k) \geq k$ [5]. □

Let k_{max} be the maximum possible k value such that a k-core of G exists. According to [6], the k-cores of graph G for all $1 \leq k \leq k_{max}$ have the following property:

Property 2.1: $\forall 1 \leq k < k_{max} : G_{k+1} \subseteq G_k$. □

Next, we define the core number for each $v \in V(G)$.

Definition 2.2: (Core Number) Given a graph G, for each node $v \in V(G)$, the core number of v, denoted by $\mathsf{core}(v, G)$, is the largest k, such that v is contained in a k-core, i.e., $\mathsf{core}(v, G) = \max\{k|v \in V(G_k)\}$. For simplicity, we use $\mathsf{core}(v)$ to denote $\mathsf{core}(v, G)$ if the context is self-evident. □

Based on Property 2.1 and Definiton 2.2, we can easily derive the following lemma:

Lemma 2.1: *Given a graph G and an integer k, let $V_k = \{v \in V(G)|\mathsf{core}(v) \geq k\}$, we have $G_k = G(V_k)$.* □

Problem Statement. In this paper, we study the problem of Core Decomposition, which is defined as follows: Given a graph G, core decomposition computes the the k-cores of G for all $1 \leq k \leq k_{max}$. We also consider how to update the k-cores of G for

all $1 \leq k \leq k_{max}$ incrementally when G is dynamically updated by insertion and deletion of edges.

According to Lemma 2.1, core decomposition is equivalent to computing $\mathsf{core}(v)$ for all $v \in V(G)$. Therefore in this paper, we study how to compute $\mathsf{core}(v)$ for all $v \in V(G)$ and how to maintain them incrementally when graph dynamically updates .

Considering that many real-world graphs are huge and cannot entirely reside in main memory, we aim to design I/O efficient algorithms to compute and maintain the core numbers of all nodes in the graph G. To analyze the algorithm, we use the external memory model introduced in [35]. Let M be the size of main memory and let B be the disk block size. A read I/O will load one block of size B from disk into main memory, and a write I/O will write one block of size B from the main memory into disk.

Assumption. In this paper, we follow a semi-external model by assuming that the nodes can be loaded in main memory while the edges are stored on disk, i.e., we assume that $M \geq c \times |V(G)|$ where c is a small constant. This assumption is practical because in most social networks and web graphs, the number of edges is much larger than the number of nodes.

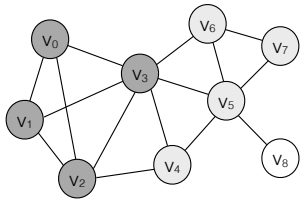


Fig. 1: A Sample Graph G and its Core Decomposition

Example 2.1: Consider the graph G in Fig. 1, the induced subgraph of $\{v_0,\ v_1,\ v_2,\ v_3\}$ is a 3-*core* in which every node has a degree at least 3. Since no 4-core exists, we have $\mathsf{core}(v_0) = \mathsf{core}(v_1) = \mathsf{core}(v_2) = \mathsf{core}(v_3) = 3$. Similarly, we can derive that $\mathsf{core}(v_4) = \mathsf{core}(v_5) = \mathsf{core}(v_6) = \mathsf{core}(v_7) = 2$ and $\mathsf{core}(v_8) = 1$. When an edge (v_7, v_8) is inserted into G, $\mathsf{core}(v_8)$ increases from 1 to 2, and the core numbers of other nodes keep unchanged. □

3 Existing Solutions

In this section, we introduce three state-of-the-art solutions for in-memory core decomposition, I/O efficient core decomposition and in-memory core maintenance respectively.

In-memory Core Decomposition. The state-of-the-art in-memory core decomposition algorithm, denote by IMCore, is proposed in [6]. The pseudocode of IMCore is shown in Algorithm 1. The algorithm processes the node with core number k in increasing order of k. Each time, k is selected as the minimum degree of current nodes in the graph (line 3). Whenever there exists a node v with degree no larger than k in the graph (line 4), we can guarantee that the core number of v is k (line 5) and we remove v with all its incident edges from the graph (line 6). Finally, the core numbers of all nodes are returned (line 7). With the help of bin sort to maintain the minimum degree of the graph, IMCore can achieve a time complexity of $O(m+n)$, which is optimal.

Algorithm 1 IMCore(Graph G)

```
1: G' ← G;
2: while G' ≠ ∅ do
3:     k ← min_{v∈V(G')} deg(v, G');
4:     while ∃v ∈ V(G') : deg(v, G') ≤ k do
5:         core(v) ← k;
6:         remove v and its incident edges from G';
7: return core(v) for all v ∈ V(G);
```

I/O Efficient Core Decomposition. The state-of-the-art efficient core decomposition algorithm is proposed in [1]. The algorithm, denoted as EMCore, is shown in Algorithm 2. It first divides the whole graph G into partitions on disk (line 1). Each partition contains a disjoint set of nodes along with their incident edges. An upper bound of $\mathsf{core}(v)$, denoted by $\mathsf{ub}(v)$, is computed for each node v in each partition P_i. Then the algorithm iteratively computes the core numbers for nodes in a top-down manner.

In iteration, the nodes with core values falling in a certain range $[k_l, k_u]$ are computed (line 6-14). Here, k_l is estimated based on the number of partitions that can be loaded in main memory (line 6). In line 7, the algorithm computes the set of partitions each of which contains at least one node v with $\mathsf{ub}(v)$ falling in $[k_l, k_u]$, and in line 8, all such partitions are loaded in main memory to form an in-memory graph G_{mem}. In line 9, an in-memory core decomposition algorithm is applied on G_{mem}, and those nodes in G_{mem} with core numbers falling in $[k_l, k_u]$ get their exact core numbers in G. After that, for all partitions loaded in memory (line 10), those nodes with exact core numbers computed are removed from the partition (line 11), and their core number upper bounds and degrees are updated accordingly (line 12). Here the new node degrees have to consider the deposited degrees from the removed nodes. Finally, the in-memory partitions are merged and written back to disk (line 13), and k_u is set to be $k_l - 1$ to process the next range of k values in the next iteration.

The I/O complexity of EMCore is $O(\frac{d \cdot (m+n)}{B})$. The CPU complexity of EMCore is $O(d \cdot (m+n))$. However, the space complexity of EMCore cannot be well bounded. In the worst case, it still requires $O(m+n)$ memory space to load the whole graph into main memory. Therefore, EMCore is not scalable to handle large-sized graphs.

Algorithm 2 EMCore(Graph G on Disk)

```
1: divide G into partitions P = {P_1, P_2, ..., P_t} on disk;
2: for all partition P_i ∈ P do
3:     compute ub(v) for all v ∈ V(P_i);
4: k_u ← +∞;
5: while k_u > 0 do
6:     estimate k_l;
7:     P_mem ← {P_i ∈ P | ∃v ∈ V(P_i) : ub(v) ∈ [k_l, k_u]};
8:     G_mem ← load partitions in P_mem in main memory;
9:     core(v) ← core(v, G_mem) for all core(v, G_mem) ∈ [k_l, k_u];
10:    for all partition P_i ∈ P_mem do
11:        remove nodes v with core(v, G_mem) ∈ [k_l, k_u] from P_i;
12:        update ub(v) and deg(v) for all v ∈ V(P_i);
13:        write P_i back to disk (merge small partitions if necessary);
14:    k_u ← k_l − 1;
15: return core(v) for all v ∈ V(G);
```

In-memory Core Maintenance. To handle the case when the graph is dynamically updated by insertion and deletion of edges, the state-of-the-art core maintenance algorithms are proposed in [2] and [3], which are based on the same findings shown in the following theorems:

Theorem 3.1: *If an edge is inserted into (deleted from) graph G, the core number $\mathsf{core}(v)$ for any $v \in V(G)$ may increase (decrease) by at most* 1. □

Theorem 3.2: *If an edge (u, v) is inserted into (deleted from) graph G, suppose $\mathsf{core}(v) \leq \mathsf{core}(u)$ and let V' be the set of nodes whose core numbers have changed, if $V' \neq \emptyset$, we have:*

- *$G(V')$ is a connected subgraph of G;*
- *$v \in V'$; and*
- *$\forall v' \in V' : \mathsf{core}(v') = \mathsf{core}(v)$;* □

Based on Theorem 3.1 and Theorem 3.2, after an edge (u, v) is inserted into (deleted from) graph G, suppose $\mathsf{core}(v) \leq \mathsf{core}(u)$,

instead of computing the core numbers for all nodes in G from scratch, we can restrain the core computation within a small range of nodes V' in G. Specifically, we can follow a two-step approach: In the first step, we can perform a depth-first-search from node v in G to compute all nodes v' with $\mathsf{core}(v') = \mathsf{core}(v)$ that are reachable from v via a path that consists of nodes with core numbers equal to $\mathsf{core}(v)$. Such nodes form a set V' which is usually much smaller than $V(G)$. In the second step, we only restrain the core number updates within the subgraph $G(V')$ in memory, and each update increases (decreases) the core number of a node by at most 1. The algorithm details and other optimization techniques can be found in [2] and [3].

4 I/O Efficient Core Decomposition

4.1 Basic Semi-external Algorithm

Drawback of EMCore. EMCore (Algorithm 2) is the state-of-the-art I/O efficient core decomposition algorithm. However, EMCore cannot be used to handle big graphs, since the number of partitions to be loaded into main memory in each iteration cannot be well-bounded. In line 7-8 of Algorithm 2, as long as a partition contains a node v with $\mathsf{ub}(v) \in [k_l, k_u]$, the whole partition needs to be loaded into main memory. When k_u becomes small, it is highly possible for a partition to contain a node v with $\mathsf{ub}(v) \in [k_l, k_u]$. Consequently, almost all partitions are loaded into main memory. Due to this reason, the space used for EMCore is $O(m+n)$, and it cannot be significantly reduced in practice.

Locality Property. In this paper, we aim to design a semi-external algorithm for core decomposition. Note that the graph is stored in disk by adjacency list. First, we introduce a locality property [36] for core numbers below:

Theorem 4.1: ***(Locality)*** *Given a vertex v, the core number $\mathsf{core}(v) = k$ if and only if:*

- *There exists a subset $V_k \subseteq \mathsf{nbr}(v)$ such that $|V_k| = \mathsf{core}(v)$ and $\forall u \in V_k : \mathsf{core}(u) \geq \mathsf{core}(v)$; and*
- *There does not exist a subset $V_{k+1} \subseteq \mathsf{nbr}(v)$ such that $|V_{k+1}| = \mathsf{core}(v) + 1$ and $\forall u \in V_{k+1} : \mathsf{core}(u) \geq \mathsf{core}(v) + 1$.* □

Based on Theorem 4.1, $\mathsf{core}(v)$ can be calculated using the following recursive equation:

$$\mathsf{core}(v) = \max k \text{ s.t. } |\{u \in \mathsf{nbr}(v) | \mathsf{core}(u) \geq k\}| \geq k \quad (1)$$

Based on Theorem 4.1, a distributed algorithm is designed in [36], in which each node v initially assigns its core number as an arbitrary core number upper bound (e.g., $\deg(v)$), and keeps updating its core numbers using Eq. 1 until convergence.

Algorithm 3 SemiCore(Graph G on Disk)

```
1: core̅(v) ← deg(v) for all v ∈ V(G);
2: update ← true;
3: while update do
4:     update ← false;
5:     for v ← v_1 to v_n do
6:         load nbr(v) from disk;
7:         c_old ← core̅(v);
8:         core̅(v) ← LocalCore(c_old, nbr(v));
9:         if core̅(v) ≠ c_old then update ← true;
10: return core̅(v) for all v ∈ V(G);

11: Procedure LocalCore(c_old, nbr(v))
12: num(i) ← 0 for all 1 ≤ i ≤ c_old;
13: for all u ∈ nbr(v) do
14:     i ← min{c_old, core̅(u)};
15:     num(i) ← num(i) + 1;
16: s ← 0;
17: for k ← c_old to 1 do
18:     s ← s + num(k);
19:     if s ≥ k then break;
20: return k;
```

Basic Solution. In this paper, we make use of the locality property to design a semi-external algorithm for core decomposition. The algorithm is shown in Algorithm 3. Here, we use $\overline{\mathsf{core}}(v)$ to denote the intermediate core number for v, which is always an upper bound of $\mathsf{core}(v)$ and will finally converge to $\mathsf{core}(v)$. Initially, $\overline{\mathsf{core}}(v)$ is assigned as an arbitrary upper bound of $\mathsf{core}(v)$ (e.g., $\deg(v)$). Then, we iteratively update $\overline{\mathsf{core}}(v)$ for all $v \in V(G)$ using the locality property until convergence (line 2-9).

In each iteration (line 5-9), we sequentially scan the node table on disk to get the offset and $\mathsf{deg}(v)$ for each node v from v_1 to v_n (line 5). Then we load $\mathsf{nbr}(v)$ from disk using the offset and $\mathsf{deg}(v)$ for each such node v, (line 6). Recall that the edge table on disk stores $\mathsf{nbr}(v)$ from v_1 to v_n sequentially. Therefore, we can load $\mathsf{nbr}(v)$ easily using sequential scan of the edge table on disk. In line 7-9, we record the original core number $\mathsf{c_{old}}$ of v (line 7); compute an updated core number of v using Eq. 1 by invoking $\mathsf{LocalCore}(\mathsf{c_{old}}, \mathsf{nbr}(v))$ (line 8); and continue the iteration if $\overline{\mathsf{core}}(v)$ is updated (line 9). Finally, when $\overline{\mathsf{core}}(v)$ for all $v \in V(G)$ keeps unchanged, we return them as their core numbers (line 10).

The procedure $\mathsf{LocalCore}(\mathsf{c_{old}}, \mathsf{nbr}(v))$ to compute the new core number of v using Eq. 1 is shown in line 11-20 of Algorithm 3. We use $\mathsf{num}(i)$ to denote the number of neighbors of v with $\overline{\mathsf{core}}$ equals i (if $i < \mathsf{c_{old}}$) or no smaller than i (if $i = \mathsf{c_{old}}$) (line 12-15). After computing $\mathsf{num}(i)$ for all $1 \leq i \leq \mathsf{c_{old}}$, we decrease k from $\mathsf{c_{old}}$ to 1 (line 17), and for each k, we compute the number of neighbors of v with $\overline{\mathsf{core}} \geq k$, denoted by s (line 18), i.e., $s = |\{u \in \mathsf{nbr}(v) | \overline{\mathsf{core}}(u) \geq k\}|$. Once $s \geq k$, we get the maximum k with $|\{u \in \mathsf{nbr}(v) | \overline{\mathsf{core}}(u) \geq k\}| \geq k$, and we return k as the new core number (line 20). Since $\mathsf{c_{old}} \leq \mathsf{nbr}(v)$, the time complexity of $\mathsf{LocalCore}(\mathsf{c_{old}}, \mathsf{nbr}(v))$ is $O(\mathsf{deg}(v))$.

Algorithm Analysis. Let l be the number of iterations of Algorithm 3. The space, CPU time, I/O complexity of Algorithm 3 are shown below (B is the block size):

Theorem 4.2: *The space, I/O, and time comlexities of Algorithm 3 are $O(n)$, $O(\frac{l \cdot (m+n)}{B})$, and $O(l \cdot (m+n))$ respectively.* □

Discussion. Note that we use a value l to denote the number of iterations of Algorithm 3. Although l is bounded by n as proved in [36], it is much smaller in practice and is usually not largely influenced by the size of the graph. For example, in a social network Twitter with $n = 41.7$ M, $m = 1.47$ G, and $k_{max} = 2488$ used in our experiments, the number of iterations using Algorithm 3 is only 62. In a web graph UK with $n = 105.9$ M, $m = 3.74$ G, and $k_{max} = 5704$ used in our experiments, the number of iterations is 2137. In the largest dataset Clueweb with $n = 978.4$ M, $m = 42.57$ G, and $k_{max} = 4244$ used in our experiments, the number of iterations is only 943.

TABLE 1: Illustration of SemiCore

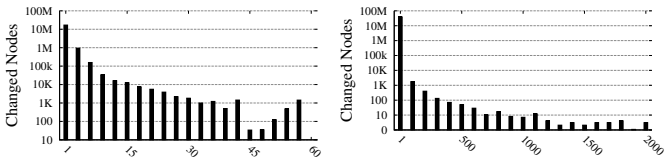
Iteration\v	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
Init	3	3	4	6	3	5	3	2	1
Iteration 1	3	3	3	3	3	3	2	2	1
Iteration 2	3	3	3	3	3	2	2	2	1
Iteration 3	3	3	3	3	2	2	2	2	1
Iteration 4	3	3	3	3	2	2	2	2	1

Example 4.1: The process to compute the core numbers for nodes in Fig. 1 using Algorithm 3 is shown in Table 1. The number in each cell is the value $\overline{\mathsf{core}}(v_i)$ for the corresponding node v_i in each iteration. The grey cells are those whose upper bounds is computed through invoking LocalCore. In iteration 1, when processing v_3, the $\overline{\mathsf{core}}$ values for the neighbors of v_3 are $\{3, 3, 3, 3, 5, 3\}$. There are 3 neighbors with $\overline{\mathsf{core}} \geq 3$ but no 4

neighbors with $\overline{\mathsf{core}} \geq 4$. Therefore, $\overline{\mathsf{core}}(v_3)$ is updated from 6 to 3. The algorithm terminates in 4 iterations. □

4.2 Optimal Node Computation

In previous subsection, for all nodes $v \in V(G)$ in each iteration of Algorithm 3, the neighbors of v are loaded from disk and $\overline{\mathsf{core}}(v)$ is recomputed. However, if we can guarantee that $\overline{\mathsf{core}}(v)$ is unchanged after recomputation, there is no need to recompute $\overline{\mathsf{core}}(v)$ by invoking LocalCore.



(a) Twitter (Vary Iteration) (b) UK (Vary Iteration)

Fig. 2: Number of Nodes Whose Core Numbers are Changed

To illustrate the effectiveness of eliminating such useless node computation, in Fig. 2, we show the number of nodes whose $\overline{\mathsf{core}}$ values are updated in each iteration for the Twitter and UK datasets used in our experiments. For the Twitter dataset, the algorithm runs for a total of 62 iterations. In iteration 1, 10 M nodes have their core numbers updated. However, in iteration 5, only 1 M nodes have their core numbers updated, which is only 10% of the number in iteration 1. From iteration 30 on, less then 2 K nodes have their core numbers updated in each iteration. For the UK dataset, the observation is similar. A total of 2137 iterations are executed. The number of core number updates in iteration 1 is 10^4 times larger than that in iteration 100, and from iteration 400 to iteration 2137, less than 100 nodes have their core numbers updated in each iteration.

The above observations indicate that reducing the number of useless node computations can largely improve the performance of the algorithm. In this section, we aim to design an optimal node computation scheme that guarantees that every execution of LocalCore will update the core number.

The Rationale. Our general idea is to maintain more node information which can be used to check whether a node computation is needed. Note that $\overline{\mathsf{core}}(v)$ for each $v \in V(G)$ will never increase, and according to Eq. 1, $\mathsf{core}(v)$ is determined by the number of neighbors u with $\mathsf{core}(u) \geq \mathsf{core}(v)$. Therefore, for each node v in the graph, we maintain the number of such neighbors, denoted by $\mathsf{cnt}(v)$, which is defined as follows:

$$\mathsf{cnt}(v) = |\{u \in \mathsf{nbr}(v) | \overline{\mathsf{core}}(u) \geq \overline{\mathsf{core}}(v)\}| \quad (2)$$

With the assistance of $\mathsf{cnt}(v)$ for all $v \in V(G)$, we can derive a sufficient and necessary condition for the core number of a node to be updated using the following lemma:

Lemma 4.1: *For each node $v \in V(G)$, $\overline{\mathsf{core}}(v)$ is updated if and only if* $\mathsf{cnt}(v) < \overline{\mathsf{core}}(v)$. □

Proof: We first prove ⇐: Suppose $\mathsf{cnt}(v) < \overline{\mathsf{core}}(v)$, we have $|\{u \in \mathsf{nbr}(v) | \overline{\mathsf{core}}(u) \geq \overline{\mathsf{core}}(v)\}| < \overline{\mathsf{core}}(v)$. Consequently, $\overline{\mathsf{core}}(v)$ needs to be decreased by at least 1 to satisfy Eq. 1.

Next, we prove ⇒: Suppose $\overline{\mathsf{core}}(v)$ needs to be updated. According to Eq. 1, either $|\{u \in \mathsf{nbr}(v) | \overline{\mathsf{core}}(u) \geq \overline{\mathsf{core}}(v)\}| < \overline{\mathsf{core}}(v)$ or there is a larger k s.t. $|\{u \in \mathsf{nbr}(v) | \overline{\mathsf{core}}(u) \geq k\}| \geq k$. The latter is impossible since $\overline{\mathsf{core}}(u)$ will never increase during the algorithm. Therefore, $\mathsf{cnt}(v) < \overline{\mathsf{core}}(v)$. □

Algorithm Design. Based on the above discussion, we propose a new algorithm SemiCore* with optimal node computation. The algorithm is shown in Algorithm 4. In line 1-3, we initialize $\overline{\mathsf{core}}(v)$, $\mathsf{active}(v)$ and update. For $\mathsf{cnt}(v)$ ($v \in V(G)$), we initialize it to be 0 which will be updated to its real value after the first iteration. In each iteration, we partially scan the graph on disk (line 6-11). Specifically, for each node v, the condition to load $\mathsf{nbr}(v)$ from disk is $\mathsf{cnt}(v) < \overline{\mathsf{core}}(v)$ according to Lemma 4.1 (line 6-8). In line 9, we compute the new $\overline{\mathsf{core}}(v)$, and we can guarantee that $\overline{\mathsf{core}}(v)$ will decrease by at least 1. In line 10, we compute $\mathsf{cnt}(v)$ by invoking $\mathsf{ComputeCnt}(\mathsf{nbr}(v), \overline{\mathsf{core}}(v))$ (line 13-17) which follows Eq. 2. In line 11, since $\overline{\mathsf{core}}(v)$ has been decreased from $\mathsf{c_{old}}$, we need to update $\mathsf{cnt}(u)$ for every $u \in \mathsf{nbr}(v)$ by invoking $\mathsf{UpdateNbrCnt}(\mathsf{nbr}(v), \mathsf{c_{old}}, \overline{\mathsf{core}}(v))$ (line 18-20). Here, according to Eq. 2, only those nodes u with $\overline{\mathsf{core}}(u)$ falling in the range $(\overline{\mathsf{core}}(v), \mathsf{c_{old}}]$ will have $\mathsf{cnt}(u)$ decreased by 1 (line 20). Finally, after the algorithm converges, the final core numbers for nodes in the graph are returned (line 12).

Algorithm 4 SemiCore* (Graph G on Disk)

```
1: core(v) ← deg(v) for all v ∈ V(G);
2: cnt(v) ← 0 for all v ∈ V(G);
3: update ← true;
4: while update do
5:     update ← false;
6:     for v ← v_1 to v_n s.t. cnt(v) < core(v) do
7:         update ← true;
8:         load nbr(v) from disk;
9:         c_old ← core(v); core(v) ← LocalCore(c_old, nbr(v));
10:        cnt(v) ← ComputeCnt(nbr(v), core(v));
11:        UpdateNbrCnt(nbr(v), c_old, core(v));
12: return core(v) for all v ∈ V(G);

13: Procedure ComputeCnt(nbr(v), core(v))
14: s ← 0;
15: for all u ∈ nbr(v) do
16:     if core(u) ≥ core(v) then s ← s + 1;
17: return s;

18: Procedure UpdateNbrCnt(nbr(v), c_old, core(v))
19: for all u ∈ nbr(v) do
20:     if core(v) < core(u) ≤ c_old then cnt(u) ← cnt(u) − 1;
```

Compared to Algorithm 3, Algorithm 4 can largely reduce the number of node computations since Algorithm 4 only computes the core number of a node whenever necessary. On the other hand, for each node v to be computed, in addition to invoking LocalCore, Algorithm 4 takes extra cost to maintain $\mathsf{cnt}(v)$ using ComputeCnt, and update $\mathsf{cnt}(u)$ for $u \in \mathsf{nbr}(v)$ using UpdateNbrCnt. However, it is easy to see that both ComputeCnt and UpdateNbrCnt take $O(\mathsf{deg}(v))$ time which is the same as the time complexity of LocalCore. Therefore, the extra cost can be well bounded.

Algorithm Analysis. Compared to the state-of-the-art I/O efficient core decomposition algorithm EMCore, SemiCore* (Algorithm 4) has the following advantages:

A_1: Bounded Memory. SemiCore* follows the semi-external model and requires only $O(n)$ memory while EMCore requires $O(m+n)$ memory in the worst case. For instance, to handle the Orkut dataset with 3 M nodes and 117.2 M edges used in our experiments, SemiCore* consumes 12 M memory; EMCore consumes 938 M memory; and the in-memory algorithm IMCore (Algorithm 1) consumes 1070 M memory.

A_2: Read I/O Only. In SemiCore*, we only require read I/Os by scanning the node and edge tables sequentially on disk in each iteration. However, EMCore needs both read and write I/Os since the partitions loaded into main memory will be repartitioned and written back to disk in each iteration. In practice, a write I/O is usually much slower than a read I/O.

A_3: Simple In-memory Structure and Data Access. In EMCore, it invokes the in-memory algorithm IMCore that uses a complex data structure for bin sort. It also involves complex graph partitioning and repartitioning algorithms. In SemiCore*, we only use two

Algorithm 5 SemiDelete*(Graph G on Disk, Edge (u, v))

1: delete (u, v) from G;
2: **if** $\overline{\mathsf{core}}(u) < \overline{\mathsf{core}}(v)$ **then**
3: $\quad \mathsf{cnt}(u) \leftarrow \mathsf{cnt}(u) - 1$;
4: **else if** $\overline{\mathsf{core}}(v) < \overline{\mathsf{core}}(u)$ **then**
5: $\quad \mathsf{cnt}(v) \leftarrow \mathsf{cnt}(v) - 1$;
6: **else**
7: $\quad \mathsf{cnt}(u) \leftarrow \mathsf{cnt}(u) - 1$; $\mathsf{cnt}(v) \leftarrow \mathsf{cnt}(v) - 1$;
8: line 3-12 of Algorithm 4;

arrays core and cnt, and the data access is simple. This makes SemiCore* very efficient in practice and even more efficient than the in-memory algorithm IMCore in many datasets. For instance, to handle the Orkut dataset used in our experiments, EMCore, IMCore, and SemiCore* consumes 63.2 seconds, 18.4 seconds, and 16.3 seconds respectively.

TABLE 2: ILLUSTRATION OF SemiCore*

Iteration\v	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
Init	3	3	4	6	3	5	3	2	1
Iteration 1	3	3	3	3	3	3	2	2	1
Iteration 2	3	3	3	3	3	2	2	2	1
Iteration 3	3	3	3	3	2	2	2	2	1

Example 4.2: The process to handle the graph G in Fig. 1 using Algorithm 4 is shown in Table 2. We show $\overline{\mathsf{core}}(v)$ each $v \in V(G)$ in each iteration, and those recomputed $\overline{\mathsf{core}}(v)$ values are shown in grey cells. For instance, after iteration 1, we have $\overline{\mathsf{core}}(v_5) = 3$ and $\mathsf{cnt}(v_5) = 2$ since only its two neighbors v_3 and v_4 have their $\overline{\mathsf{core}}$ values no smaller than 3. Therefore, in iteration 2, $\overline{\mathsf{core}}(v_5)$ is recomputed and updated from 3 to 2. This also updates the cnt value of its neighbor v_4 from 3 to 2 since $\overline{\mathsf{core}}(v_4) = 3$. Note that in iteration 1, we need to compute $\overline{\mathsf{core}}(v)$ for all $v \in V(G)$ since $\mathsf{cnt}(v)$ is unknown only in the first iteration. Compared to Algorithm 3 in Example 4.1, Algorithm 4 only uses 3 iterations and reduces the number of node computations from 23 to 11. □

5 I/O EFFICIENT CORE MAINTENANCE

In this section, we discuss how to incrementally maintain the core numbers when edges are inserted into or deleted from the graph under the semi-external setting.

5.1 Edge Deletion

Algorithm Design. From Theorem 3.1, we know that after an edge deletion, the core number for any $v \in V(G)$ will decrease by at most 1. Therefore, after an edge deletion, the old core numbers of nodes in the graph are upper bounds of their new core numbers. Recall that in Algorithm 4, as long as $\overline{\mathsf{core}}(v)$ is initialized to be an arbitrary upper bound of $\mathsf{core}(v)$ for all $v \in V(G)$, $\overline{\mathsf{core}}(v)$ can be converged to $\mathsf{core}(v)$ after the algorithm terminates. Therefore, Algorithm 4 can be easily modified to handle edge deletion.

Specifically, we show our algorithm SemiDelete* for edge deletion in Algorithm 5. Given an edge $(u, v) \in E(G)$ to be removed, we first delete (u, v) from G (line 1). In line 2-7, we update $\mathsf{cnt}(u)$ and $\mathsf{cnt}(v)$ due to the deletion of edge (u, v). Here, we consider three cases. First, if $\overline{\mathsf{core}}(u) < \overline{\mathsf{core}}(v)$, we only need to decrease $\mathsf{cnt}(u)$ by 1. Second, if $\overline{\mathsf{core}}(v) < \overline{\mathsf{core}}(u)$, we decrease $\mathsf{cnt}(v)$ by 1. Third, if $\overline{\mathsf{core}}(v) = \overline{\mathsf{core}}(u)$, we decrease both $\mathsf{cnt}(v)$ and $\mathsf{cnt}(u)$ by 1. Now we can use Algorithm 4 to update the core numbers of other nodes (line 8).

Example 5.1: Suppose after Example 4.2, we delete edge (v_0, v_1) from G (Fig. 1). By Algorithm 5, we update $\mathsf{cnt}(v_0)$ and $\mathsf{cnt}(v_1)$ from 3 to 2 and invoke line 3-12 of Algorithm 4. Only 1 iteration is needed with 4 node computations as shown in Table 3. □

TABLE 3: ILLUSTRATION OF SemiDelete* (DELETE (v_0, v_1))

Iteration\v	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
Old Value	3	3	3	3	2	2	2	2	1
Iteration 1	2	2	2	2	2	2	2	2	1

Algorithm Analysis. Algorithm 5 performs the same operations as Algorithm 4. Therefore, it can be obviously bounded by the same space, CPU time and I/O complexities as in Theorem 4.2. For a removed edge (u, v), without loss of generality, we assume $\mathsf{core}(u) \leq \mathsf{core}(v)$. Let l be the number of iterations of Algorithm 5 and V_c be the set of nodes reachable from u via a path that consists of nodes w with $\mathsf{core}(w) = \mathsf{core}(u)$, we have:

Theorem 5.1: *The space, I/O, and time complexities of Algorithm 5 are $O(n)$, $O(\frac{l \cdot (m+n)}{B})$, and $O(\sum_{w \in V_c} \mathsf{deg}(w) + l \cdot n)$ respectively.* □

The iteration number l for Algorithm 5 is bounded by $|V_c|$, which is much smaller than n in practice.

5.2 Edge Insertion

The Rationale. After a new edge (u, v) is inserted into graph G, according to Theorem 3.1, we know that the core number for any $v \in V(G)$ will increase by at most 1. As a result, the old core number of a node in the graph may not be an upper bound of its new core number. Therefore, Algorithm 4 cannot be applied directly to handle edge insertion. However, according to Theorem 3.2, after inserting an edge (u, v) (suppose $\overline{\mathsf{core}}(v) \leq \overline{\mathsf{core}}(u)$), we can find a candidate set V_c consisting of all nodes w that are reachable from node v via a path that consists of nodes with $\overline{\mathsf{core}}$ equals $\overline{\mathsf{core}}(v)$, and we can guarantee that those nodes with core numbers increased by 1 is a subset of V_c. Consequently, if we increase $\overline{\mathsf{core}}(v)$ by 1 for all $v \in V_c$, we can guarantee that for all $u \in V(G)$, $\overline{\mathsf{core}}(u)$ is an upper bound of the new core number of u. Thus we can apply Algorithm 4 to compute the new core numbers.

TABLE 4: ILLUSTRATION OF SemiInsert (INSERT (v_4, v_6))

Iteration\v	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
Old Value	2	2	2	2	2	2	2	2	1
Iteration 1.1	2	2	2	2	3	3	3	3	1
Iteration 1.2	2	2	3	3	3	3	3	3	1
Iteration 1.3	3	3	3	3	3	3	3	3	1
Iteration 2.1	2	2	2	3	3	3	3	2	1

Algorithm Design. Our algorithm SemiInsert for edge insertion is shown in Algorithm 6. In line 1, we insert (u, v) into G. In line 1-4, we update $\mathsf{cnt}(u)$ and $\mathsf{cnt}(v)$ caused by the insertion of edge (u, v). We use $\mathsf{active}(w)$ to denote whether w is a candidate node with core number increased which is initialized to be false except for node u. In line 8-20, we iteratively update $\mathsf{active}(w)$ for $w \in V(G)$ until convergence. In each iteration (line 10-20), we find nodes v' with $\mathsf{active}(v') =$ true and $\overline{\mathsf{core}}(v')$ not being increased (line 11). For each such node v', we increase $\overline{\mathsf{core}}(v')$ by 1 (line 13), and load $\mathsf{nbr}(v')$ from disk. Since $\overline{\mathsf{core}}(v')$ is changed, we need to compute $\mathsf{cnt}(v')$ (line 15) and update the cnt values for the neighbors of v' (line 16-17). In line 18-20, we set $\mathsf{active}(u')$ to be true for all the neighbors u' of v' if u' is a possible candidate. Now we can guarantee that $\overline{\mathsf{core}}(v')$ is an upper bound of the new core number of v'. Therefore, we invoke line 3-12 of Algorithm 4 to compute the core numbers of all nodes in the graph (line 21).

Example 5.2: Suppose after deleting edge (v_0, v_1) from the graph G (Fig. 1) in Example 5.1, we insert a new edge (v_4, v_6) into G.

The process to compute the new core numbers of nodes in G is shown in Table 4. Here, we use 3 iterations 1.1, 1.2, and 1.3 to compute the candidate nodes, and use 1 iteration 2.1 to compute the new core numbers. In iteration 1.1, when v_4 is computed, it triggers its smaller neighbors v_2 and v_3 to be computed in the next iteration and triggers its larger neighbor v_5 to be computed in the current iteration. The total number of node computations is 12. □

Algorithm 6 SemiInsert(Graph G on Disk, Edge (u, v))

```
1: insert (u, v) into G;
2: swap u and v if core(u) > core(v);
3: cnt(u) ← cnt(u) + 1;
4: if core(v) = core(u) then cnt(v) ← cnt(v) + 1;
5: c_old ← core(u);
6: active(w) ← false for all w ∈ V(G); active(u) ← true;
7: update ← true;
8: while update do
9:     update ← false;
10:    for v' ← v_1 to v_n do
11:        if active(v') = true and core(v') = c_old then
12:            update ← true
13:            core(v') ← core(v') + 1;
14:            load nbr(v') from disk;
15:            cnt(v') ← ComputeCnt(nbr(v'), core(v'));
16:            for all u' ∈ nbr(v') s.t. core(u') = core(v') do
17:                cnt(u') ← cnt(u') + 1;
18:            for all u' ∈ nbr(v') do
19:                if core(u') = c_old and active(u') = false then
20:                    active(u') ← true;
21: line 3-12 of Algorithm 4;
```

Algorithm Analysis. The first phase of Algorithm 6 (line 8-20) locates all candidate nodes in V_c. The second phase of Algorithm 6 (line 21) performs the same operations as Algorithm 4. Let l_1 and l_2 be the number of iterations in the first and second phases respectively, we have:

Theorem 5.2: *Algorithm 6 requires $O(n)$ memory. The I/O and CPU time complexities of Algorithm 6 are $O(\frac{(l_1+l_2)\cdot(m+n)}{B})$ and$O(\sum_{w\in V_c}\mathsf{deg}(w) + (l_1 + l_2)\cdot n)$ respectively.* □

Here, both l_1 and l_2 are bounded by $|V_c|$, which is much smaller than n in practice.

5.3 Optimization for Edge Insertion

The Rationale. Algorithm 6 handles an edge insertion using two phases. In phase 1, we compute a superset V_c of nodes whose core numbers will be updated, and we increase the core numbers for all nodes in V_c by 1. In phase 2, we compute the core numbers of all nodes using Algorithm 4. One problem of Algorithm 6 is that the size of V_c can be very large, which may result in a large number of node computations and I/Os in both phase 1 and phase 2 of Algorithm 6. Therefore, it is crucial to reduce the size of V_c.

Now, suppose and edge (u, v) is inserted into the graph G; $\mathsf{cnt}(u)$ and $\mathsf{cnt}(v)$ are updated accordingly; and $\overline{\mathsf{core}}(w)$ values for all $w \in V(G)$ have not been updated . Without loss of generality, we assume that $\overline{\mathsf{core}}(u) < \overline{\mathsf{core}}(v)$ and let $\mathsf{c_{old}} = \overline{\mathsf{core}}(u)$. Let V_c be the set of candidate nodes computed in Algorithm 6, i.e., V_c consists of all nodes that are reachable from u via a path that consists of nodes with $\overline{\mathsf{core}}$ equals $\mathsf{c_{old}}$. Let $V_c^* \subseteq V_c$ be the set of nodes with $\overline{\mathsf{core}}$ updated to be $\mathsf{c_{old}}+1$ after inserting (u, v). We have the following lemmas:

Lemma 5.1: *(a) For $v' \in V_c \setminus V_c^*$, $\mathsf{cnt}(v')$ keeps unchanged; (b) For $v' \in V_c^*$, $\mathsf{cnt}(v')$ will not increase.* □

Proof: This lemma can be easily verified according to Eq. 2 and Theorem 3.1. □

Lemma 5.2: *If $\mathsf{cnt}(v') \geq \mathsf{c_{old}} + 1$ for all $v' \in V_c$, then we have $V_c^* = V_c$.* □

Proof: If we increase $\overline{\mathsf{core}}(v')$ by 1 for all $v' \in V_c$, it is easy to verify that $\mathsf{cnt}(v')$ for all $v' \in V_c$ keep unchanged. Now suppose $\mathsf{cnt}(v') \geq \mathsf{c_{old}} + 1$ for all $v' \in V_c$, we can derive that the locality property in Theorem 4.1 holds for every $v' \in V(G)$. Therefore, the new $\overline{\mathsf{core}}(v')$ is the core number of v' for every $v' \in V(G)$. This indicates that $V_c^* = V_c$. □

Lemma 5.3: *For any $v' \in V_c$, if $v' \in V_c^*$, then we have $\mathsf{cnt}(v') \geq \mathsf{c_{old}} + 1$.* □

Proof: Since $v' \in V_c^*$, we know that the new $\mathsf{cnt}(v')$ is no smaller than $\mathsf{c_{old}} + 1$. According to Lemma 5.1 (b), the original $\mathsf{cnt}(v')$ is also no smaller than $\mathsf{c_{old}} + 1$, since $\mathsf{cnt}(v')$ will not increase. Therefore, the lemma holds. □

Theorem 5.3: *For each $v' \in V_c$, we define $\mathsf{cnt}^*(v')$ as:*

$$\mathsf{cnt}^*(v') = |\{u' \in \mathsf{nbr}(v') \mid \overline{\mathsf{core}}(u') > \mathsf{c_{old}} \text{ or } u' \in V_c^*\}| \quad (3)$$

We have:

(a) If $v' \in V_c^*$, then the updated $\mathsf{cnt}(v') = \mathsf{cnt}^*(v')$; and

(b) $v' \in V_c^* \Leftrightarrow \mathsf{cnt}^*(v') \geq \mathsf{c_{old}} + 1$. □

Proof: For (a): for all $v' \in V_c^*$, since $\overline{\mathsf{core}}(v')$ will become $\mathsf{c_{old}} + 1$, all nodes $u' \in V_c \setminus V_c^*$ will not contribute to $\mathsf{cnt}(v')$ according to Eq. 2. Therefore, (a) holds.

For (b): $\Rightarrow$ can be derived according to (a). Now we prove $\Leftarrow$. Suppose $\mathsf{cnt}^*(v') \geq \mathsf{c_{old}} + 1$, to prove $v' \in V_c^*$, we prove that if we increase $\overline{\mathsf{core}}(u')$ to $\mathsf{c_{old}} + 1$ for all $u' \in V_c$ and apply Algorithm 4, then $\overline{\mathsf{core}}(v')$ will keep to be $\mathsf{c_{old}} + 1$ after convergence. Note that for all nodes $u' \in V_c^*$ and $u' \in \mathsf{nbr}(v')$, $\overline{\mathsf{core}}(u')$ will keep to be $\mathsf{c_{old}} + 1$ and will contribute to $\mathsf{cnt}(v')$, and all nodes $u' \in \mathsf{nbr}(v')$ with $\overline{\mathsf{core}}(u') > \mathsf{c_{old}}$ will also contribute to $\mathsf{cnt}(v')$. According to Eq. 3, we have $\mathsf{cnt}(v') \geq \mathsf{cnt}^*(v') \geq \mathsf{c_{old}} + 1$. Therefore, $\overline{\mathsf{core}}(v')$ will never decrease according to Lemma 4.1. This indicates that $v' \in V_c^*$. □

According to Theorem 5.3 (b), $\mathsf{cnt}^*(v')$ can be defined using the following recursive equation:

$$\mathsf{cnt}^*(v') = |\{u' \in \mathsf{nbr}(v') \mid \overline{\mathsf{core}}(u') > \mathsf{c_{old}} \text{ or } (\overline{\mathsf{core}}(u') = \mathsf{c_{old}} \text{ and } \mathsf{cnt}^*(u') \geq \mathsf{c_{old}} + 1)\}| \quad (4)$$

To compute $\mathsf{cnt}^*(v')$ for all $v' \in V_c$, we can initialize $\mathsf{cnt}^*(v')$ to be $\mathsf{cnt}(v')$, and apply Eq. 4 iteratively on all $v' \in V_c$ until convergence. However, this algorithm needs to compute V_c first, which is inefficient. Note that according to Eq. 4 and Theorem 5.3 (b), we only care about those nodes u' with $\mathsf{cnt}^*(u') \geq \mathsf{c_{old}} + 1$. Therefore, we do not need to compute the whole V_c by expanding from node u. Instead, for each expanded node u', if we guarantee that $\mathsf{cnt}^*(u') < \mathsf{c_{old}} + 1$, we do not need to expand u' further. In this way, the computational and I/O cost can be largely reduced.

Algorithm Design. Based on the above discussion, for each node $w \in V(G)$, we use $\mathsf{status}(w)$ to denote the status of node w during the processing of node expansion. Each node $w \in V(G)$ has the following four status ($\mathsf{status}(w)$):

- ϕ: w has not been expanded by other nodes.
- ?: w is expanded but $\mathsf{cnt}^*(w)$ is not calculated.
- ✓: $\mathsf{cnt}^*(w)$ is calculated with $\mathsf{cnt}^*(w) \geq \mathsf{c_{old}} + 1$.
- ⊗: $\mathsf{cnt}^*(w)$ is calculated with $\mathsf{cnt}^*(w) < \mathsf{c_{old}} + 1$.

With $\mathsf{status}(w)$ and according to Theorem 5.3 (a) and Lemma 5.1 (a), we can reuse $\mathsf{cnt}^*(w)$ to calculate $\mathsf{cnt}(w)$ for each $w \in V(G)$. That is, if $\mathsf{status}(w) =$ ✓, $\mathsf{cnt}^*(w)$ can directly represent $\mathsf{cnt}(w)$ according to Eq. 4, otherwise, if $\mathsf{status}(w) =$ ⊗, $\mathsf{cnt}(w)$ is calculated using Eq. 2.

Our new algorithm $\mathsf{SemiInsert}^*$ for edge insertion is shown in Algorithm 7. The initialization phase is similar to that in Algorithm 6 (line 1). In line 2, we initialize $\mathsf{status}(w)$ to be ϕ except $\mathsf{status}(u)$ which is initialized to be ?. The algorithm iteratively update $\mathsf{status}(v')$, $\overline{\mathsf{core}}(v')$, and $\mathsf{cnt}(v')$ for all $v' \in V(G)$. For

Algorithm 7 SemiInsert*(Graph G on Disk, Edge (u, v))

```
1: line 1-5 of Algorithm 6;
2: status(w) ← ϕ for all w ∈ V(G); status(u) ← ?;
3: update ← true;
4: while update do
5:     update ← false;
6:     for v' ← v_1 to v_n do
7:         if status(v') = ? then
8:             update ← true;
9:             load nbr(v') from disk;
10:            cnt(v') ← ComputeCnt*(nbr(v'), c_old);
11:            status(v') ← √; core(v') ← c_old + 1;
12:            for all u' ∈ nbr(v') s.t. core(u') = c_old + 1 do
13:                cnt(u') ← cnt(u') + 1;
14:            if cnt(v') ≥ c_old + 1 then
15:                for all u' ∈ nbr(v') s.t. core(u') = c_old do
16:                    if cnt(u') ≥ c_old + 1 and status(u') = ϕ then
17:                        status(u') ← ?;
18:        if status(v') = √ and cnt(v') < c_old + 1 then
19:            update ← true;
20:            load nbr(v') from disk if not loaded;
21:            cnt(v') ← ComputeCnt*(nbr(v'), c_old);
22:            status(v') ← ⊗; core(v') ← c_old;
23:            for all u' ∈ nbr(v') s.t. core(u') = c_old + 1 do
24:                cnt(u') ← cnt(u') − 1;
25:            for all u' ∈ nbr(v') s.t. status(u') = √ do
26:                cnt(u') ← cnt(u') − 1;
27: Procedure ComputeCnt*(nbr(v'), c_old)
28: s ← 0;
29: for all u' ∈ nbr(v') do
30:     if core(u') > c_old or (core(u') = c_old and cnt(u') ≥ c_old + 1 and
        status(u') ≠ ⊗) then s ← s + 1;
31: return s;
```

each such v' to be checked in every iteration (line 4-26), we consider the following status transitions:

• *From ? to √ (line 7-13):* If $\mathsf{status}(v') = ?$ (line 7), we load $\mathsf{nbr}(v')$ from disk (line 9) and compute $\mathsf{cnt}(v')$ using Eq. 4 by invoking $\mathsf{ComputeCnt}^*(\mathsf{nbr}(v'), \mathsf{c_{old}})$ which is shown in line 27-31. Compared to Eq. 4, we add a new condition for $u' \in \mathsf{nbr}(v')$: $\mathsf{status}(u') \neq \otimes$ (line 30). This is because for node u' with $\mathsf{status}(u') = \otimes$, it is computed using Eq. 2 other than Eq. 4, and it cannot contribute to $\mathsf{cnt}(v')$. After computing $\mathsf{cnt}(v')$, in line 11, we set $\mathsf{status}(v')$ to be $\surd$ and increase $\overline{\mathsf{core}}(v')$ to be $\mathsf{c_{old}} + 1$. Since $\overline{\mathsf{core}}(v')$ is increased to be $\mathsf{c_{old}} + 1$, we need to increase $\mathsf{cnt}(u')$ for all neighbor u' of v' with $\overline{\mathsf{core}}(u') = \mathsf{c_{old}} + 1$ (line 12-13).

• *From ϕ to ? (line 14-17):* After setting v' to be ?, if $\mathsf{cnt}(v') \geq \mathsf{c_{old}} + 1$, v' will not set to be $\otimes$ in this iteration. In this case (line 14), we can expand v'. That is, for all neighbors u' of v' with $\overline{\mathsf{core}}(u') = \mathsf{c_{old}}$ (line 15), if $\mathsf{cnt}(u') \geq \mathsf{c_{old}} + 1$ (refer to Lemma 5.3) and u' has not be expanded ($\mathsf{status}(u') = \phi$), we set $\mathsf{status}(u')$ to be ? so that u' can be expanded.

• *From √ to ⊗ (line 18-26):* If $\mathsf{status}(v')$ is $\surd$ and $\mathsf{cnt}(v') < \mathsf{c_{old}} + 1$, we need to change the status of v' (line 18). Here, in line 20, we load $\mathsf{nbr}(v')$ from disk if it is not loaded in line 9. In line 21, we compute $\mathsf{cnt}(v')$ using Eq. 2. In line 22, we set $\mathsf{status}(v')$ to be $\otimes$, and update $\overline{\mathsf{core}}(v')$ to be $\mathsf{c_{old}}$ according to Lemma 5.1 (a). Since $\overline{\mathsf{core}}(v')$ is changed from $\mathsf{c_{old}} + 1$ to $\mathsf{c_{old}}$, for all neighbors u' of v' with $\overline{\mathsf{core}}(u') = \mathsf{c_{old}} + 1$, we need to decrease $\mathsf{cnt}(u')$ (line 23-24). In addition, according to Eq. 4, the status change from $\surd$ to $\otimes$ for v' will trigger each neighbor u' of v' to decrease its $\mathsf{cnt}(u')$ if $\mathsf{status}(u') = \surd$ (line 25-26).

Example 5.3: Suppose after deleting edge (v_0, v_1) from graph G (Fig. 1) in Example 5.1, we insert edge (v_4, v_6) into G. The process to update the status of nodes in each iteration is shown in Table 5. In iteration 1, when we check v_4, we update $\mathsf{status}(v_4)$ from ? to be $\surd$, and update the status of its neighbors (v_2, v_3, v_5, and v_6) to be ?. In iteration 2, for v_2 with status ?, we can calculate that $\mathsf{cnt}(v_2) = 2 < \mathsf{c_{old}} + 1 = 3$. Therefore, we set $\mathsf{status}(v_2)$ to be $\otimes$, and decrease $\mathsf{cnt}(v_4)$ accordingly. The cells involving a node computation are marked grey. Totally 2 iterations are needed. The four nodes v_3, v_4, v_5, and v_6 with status being $\surd$ have their core numbers updated. Compared to Example 5.2, we decrease the number of node computations from 12 to 5. □

TABLE 5: ILLUSTRATION OF SemiInsert* (INSERT (v_4, v_6))

Iteration\v	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
Old Value	2	2	2	2	2	2	2	2	1
Iteration 1	ϕ	ϕ	?	?	√	√	√	ϕ	ϕ
Iteration 2	ϕ	ϕ	⊗	√	√	√	√	ϕ	ϕ
New Value	2	2	2	3	3	3	3	2	1

Algorithm Analysis. Theoretically, Algorithm 7 is bounded by the same space, CPU time and I/O complexities as Algorithm 6. However, compared to Algorithm 6 that requires two phases to update the core numbers, Algorithm 7 requires only one phase and the number of candidate nodes is largely reduced in Algorithm 7

6 I/O EFFICIENT DEGENERACY ORDERING

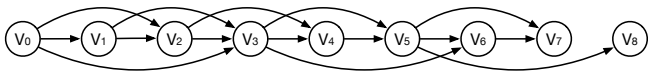
In this section, we study the problem of degeneracy ordering, which is closely related to the core decomposition problem. Given an arbitrary total order $f()$ for the nodes in graph G, let $\vec{\mathsf{deg}}(u)$ be the backward-degree, i.e., $\vec{\mathsf{deg}}(u) = |\{v | v \in \mathsf{nbr}(u), f(v) > f(u)\}|$. The degeneracy of the order is the maximum $\vec{\mathsf{deg}}(u)$ for all u. The definition of graph degeneracy is given as follows:

Definition 6.1: (Graph Degeneracy [30]) The degeneracy of a graph G, denoted by $d(G)$, is the minimum degeneracy of any total order of nodes in G. □

Definition 6.2: (Degeneracy Order [30]) A total order of nodes in G is a degeneracy order if $\forall u \in V, \vec{\mathsf{deg}}(u) \leq d(G)$. □

Example 6.1: Fig. 3 shows a degeneracy order of nodes in graph G of Fig. 1 from left to right. The corresponding backward-degree of each node is given in the table. The degeneracy of G is 3. We consider node v_2. The neighbors of node v_2 in G are v_0, v_1, v_3 and v_4. In the shown order, node v_2 has 2 backward-neighbors, v_3 and v_4. Thus the backward-degree here for v_2 is 2. □

Fig. 3: A DEGENERACY ORDER OF NODES IN FIG. 3



Order	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
Backward-Deg	3	2	2	3	1	3	1	0	0

An in-memory algorithm for the computation of the degeneracy order is proposed in [31]. The algorithm is similar to the in-memory algorithm for core decomposition [6]. Specifically, it iteratively removes the node with minimum degree in the graph until all nodes are removed. After removing a node, the degrees of other nodes are updated accordingly. The sequence of the removed nodes forms the degeneracy order, and the maximum backward-degree of nodes in the sequence is the degeneracy of graph G. It costs $O(m)$ time and $O(m)$ space. Since we follow the same process of in-memory core decomposition to compute the degeneracy order, we can derive that the graph degeneracy is equal to the max core value, i.e. k_{max}, in the graph [30]:

$$d(G) = k_{max} \tag{5}$$

In this paper, we handle the scenario that the graph is too big to load in the memory. Note that the in-memory core decomposition algorithm [6] follows a bottom-up strategy and the sequence of

removed nodes naturally forms a degeneracy order according to [31]. By contrast, in Algorithm 4, we follow a top-down strategy which iteratively updates the core numbers of all nodes until convergence. Thus we cannot directly obtain the degeneracy order as a byproduct of Algorithm 4.

6.1 Degeneracy Order Computation

We aim to compute the degeneracy order on big graphs. A straightforward semi-external algorithm can be easily obtained by following the idea of [31], which iteratively removes the nodes with the minimum degree and terminates once all nodes are removed. The algorithm is given in Algorithm 8.

Algorithm 8 Dorder(Graph G on Disk)

```
1: D ← ∅, Deg(v) ← deg(v) for all v ∈ V(G); d_min ← −1;
2: while |D| < |V| do
3:     d_min ← max(d_min, min_{u∈V\D} Deg(u));
4:     for v ← v_1 to v_n do
5:         if v ∈ D or Deg(v) > d_min then continue;
6:         D.append(v);
7:         load nbr(v) from disk;
8:         for all u ∈ nbr(v) do Deg(u) ← Deg(u) − 1;
9: return D;
```

The Rationale. Obviously, Algorithm 8 is inefficient since in each iteration it can only process the nodes with the minimum degree. Note that by performing the semi-external core decomposition algorithm, we can compute the degeneracy of the graph as $d(G) = \max_{v \in V(G)} \mathsf{core}(v)$. With $d(G)$, we can remove more nodes then just removing the nodes with the minimum degree in each iteration. Specifically, according to Definiton 6.2, we remove all nodes with degree no larger than $d(G)$ in each iteration, update the degree of the remaining nodes, and terminate once all nodes are removed. Since the degree of each removed node is less then $d(G)$, the sequence of the removed nodes forms the degeneracy order.

Algorithm 9 Dorder*(Graph G on Disk)

```
1: invoke Algorithm 4 to compute core(v) for all v ∈ V(G);
2: d(G) ← max_{v∈V(G)} core(v);
3: D ← ∅;
4: Deg(v) ← deg(v) for all v ∈ V(G);
5: while |D| < |V| do
6:     for v ← v_1 to v_n do
7:         if v ∈ D or Deg(v) > d(G) then continue;
8:         D.append(v);
9:         load nbr(v) from disk;
10:        for all u ∈ nbr(v) do Deg(u) ← Deg(u) − 1;
11: return D;
```

Algorithm Design. The algorithm Dorder* is shown in Algorithm 9. We invoke the SemiCore* algorithm for graph G in advance to calculate the core number of each node (line 1). We select the maximum core number as $d(G)$ (line 2). We use an ordered list $\mathcal{D}$ to keep the degeneracy order of all nodes (line 3), and we use $Deg(u)$ to maintain the degree of each node $u \in V(G)$ (line 4). In each iteration, we sequentially scan the nodes. For each node u such that u is not added to $\mathcal{D}$ and $Deg(u) \leq d$, we append it to the end of the degeneracy order $\mathcal{D}$ (line 8) and decrease the degree of all neighbors of u by 1 (line 10). The algorithm terminates once all nodes are added to $\mathcal{D}$.

Algorithm Analysis. Let l be the number of iterations of Algorithm 9, we have the following theorem:

Theorem 6.1: *The space, I/O, and time complexities of Algorithm 9 are $O(n)$, $O(\frac{l \cdot (m+n)}{B})$, and $O(m)$ respectively.* □

Here, l can be bounded by n in the worst case. However, it is much smaller in practice. In our experiments, the number of iterations is less than 5 on most of our tested datasets. Note that the cost to invoke Algorithm 4 is not included in Theorem 6.1.

6.2 Degeneracy Order Maintenance

In this section, we consider the scenario when the graph updates dynamically. We aim to design a semi-external algorithm to maintain the degeneracy order incrementally. We only focus on edge update since the node update can be implemented by a series of edge updates. Note that a special case here is that graph degeneracy $d(G)$ changes when the graph updates. We just recompute the degeneracy order for this case. Thus for each inserted or removed edge, we first invoke our algorithm SemiInsert* to obtain the maximum core number k_{max} and check whether $d(G)$ changes using Eq. 5. In the following, we assume that the graph degeneracy does not change when the graph updates.

The Rationale. The main challenge of degeneracy order maintenance is that we need to maintain a total order of nodes in the graph every time the graph updates. From Algorithm 9, we observe that in each iteration, there are multiple nodes u with $Deg(u) < d(G)$. It is easy to see that we can remove these nodes in an arbitrary order in the iteration to generate the degeneracy order. We can simply put these nodes or any subset of these nodes in a certain level and thus divide the nodes of the graph into several levels. With the levels, instead of maintaining the total order of nodes, we can just maintain the node levels to update the degeneracy order.

Level-Index. We propose a Level-Index which contains the following two components:

(i) The $level\text{-}value$ for each node u is a value $\mathcal{L}(u)$ satisfying:

$$|\{v \in \mathsf{nbr}(u) | \mathcal{L}(v) \geq \mathcal{L}(u)\}| \leq d(G) \tag{6}$$

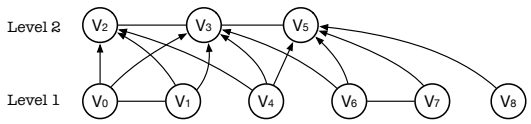
(ii) The $upper\text{-}degree$ for each node u is the number of its neighbors v whose $level\text{-}value$ is not less than that of u:

$$Deg_{\mathcal{L}}(u) = |\{v \in \mathsf{nbr}(u) | \mathcal{L}(v) \geq \mathcal{L}(u)\}| \tag{7}$$

Example 6.2: An example of Level-Index for graph G in Fig. 1 is given in Fig. 4. There exists only 2 levels for all nodes in G. The edges connecting two nodes in the same level are shown in undirected lines and those connecting the nodes in different levels are shown in directed lines. The $level\text{-}value$ and $upper\text{-}degree$ for all nodes are presented in the table. We can see that the $upper\text{-}degree$ of any node is not larger than $d(G) = 3$. We consider the node v_0. There are 3 neighbors of v_0 which has 3 neighbors, v_1, v_2 and v_3. Given $\mathcal{L}(v_0) = \mathcal{L}(v_1) = 1$ and $\mathcal{L}(v_2) = \mathcal{L}(v_3) = 2$, we have $Deg_{\mathcal{L}}(v_0) = 3$. For the node v_2, there are 4 neighbors, v_0, v_1, v_3 and v_4. Since $\mathcal{L}(v_0) = \mathcal{L}(v_1) = \mathcal{L}(v_4) = 1$ and $\mathcal{L}(v_2) = \mathcal{L}(v_3) = 2$, we have $Deg_{\mathcal{L}}(v_2) = 1$. □

Given a Level-Index, we can obtain a degeneracy order as follows. We sort all nodes in non-decreasing order of their level values, and for those node with the same level value, we sort them in an arbitrary order. It is easy to see that any node order generated in this way is a degeneracy order. Therefore, instead of maintaining the degeneracy order, we only need to maintain the Level-Index. In the following, we first show how to construct the Level-Index, and then introduce how to maintain the Level-Index when an edge is removed or inserted.

Fig. 4: A LEVEL-INDEX FOR GRAPH G IN FIG. 1



Algorithm Design for Level-Index Construction. The algorithm to construct the Level-Index is given in Algorithm 10. We follow the semi-external model. The $\mathcal{L}$ and $Deg_{\mathcal{L}}$ for all nodes are initialized by -1 (line 1) and original degree (line 2) respectively.

Level-Index	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
$\mathcal{L}$	1	1	2	2	1	2	1	1	1
$Deg_{\mathcal{L}}$	3	3	1	2	3	1	3	2	1

The $level$ is initialized by -1 (line 3). In each iteration, we assign the level of all nodes whose degree are not larger than the $d(G)$ by current $level$ (line 6 and 7). The current degree of them are their corresponding $upper\text{-}degree$. Then we remove such nodes and update the degree of their neighbors (line 9-12). We increase the $level$ (line 13) and move to next iterations. The algorithm terminates until the $level\text{-}values$ of all nodes in G are assigned.

Algorithm 10 Level-IndexConstruction$(G, \mathcal{L}, Deg_{\mathcal{L}})$

1: $\mathcal{L}(v) \leftarrow -1$ for all $v \in V(G)$;
2: $Deg_{\mathcal{L}}(v) \leftarrow \mathsf{deg}(v)$ for all $v \in V(G)$;
3: $level \leftarrow 1$;
4: **while** there exists a node u such that $\mathcal{L}(u) = -1$ **do**
5: **for** $v \leftarrow v_1$ **to** v_n **do**
6: **if** $\mathcal{L}(v) \neq -1$ or $Deg_{\mathcal{L}}(v) > d(G)$ **then continue**;
7: $\mathcal{L}(v) \leftarrow level$;
8: **for** $v \leftarrow v_1$ **to** v_n **do**
9: **if** $\mathcal{L}(v) \neq level$ **then continue**;
10: load $\mathsf{nbr}(v)$ from disk;
11: **for** $u \in \mathsf{nbr}(v)$ **do**
12: **if** $\mathcal{L}(v) = -1$ **then** $Deg_{\mathcal{L}}(u) \leftarrow Deg_{\mathcal{L}}(u) - 1$;
13: $level \leftarrow level + 1$;

Based on Eq. 6 and Eq. 7, we can derive the following sufficient and necessary condition for $\mathcal{L}$ to be updated.

Lemma 6.1: *For each node $v' \in V(G)$, $\mathcal{L}(v')$ is updated if and only if $Deg_{\mathcal{L}}(v') > d(G)$.* □

Proof: We first prove $\Leftarrow$: Suppose $Deg_{\mathcal{L}}(v') > d(G)$, we have $\mathsf{deg}(v') > d(G)$ after removing all nodes u with $\mathcal{L}(u') < \mathcal{L}(v')$. This contradicts the definition of $level\text{-}value$ in Level-Index. Therefore, $\mathcal{L}(v')$ needs to be updated.

Next, we prove $\Rightarrow$: Suppose $Deg_{\mathcal{L}}(v')$ needs to be updated, according to the definition of Level-Index, either $|\{u' \in \mathsf{nbr}(v') | \mathcal{L}(u') \geq \mathcal{L}(v')\}| > d(G)$ or there is a smaller l s.t. $\mathcal{L}(v') = l$ and $|\{u' \in \mathsf{nbr}(v') | \mathcal{L}(u') \geq \mathcal{L}(v')\}| \leq d(G)$. The latter is impossible since $\mathcal{L}(v')$ will never decrease when $d(G)$ does not change. Therefore, we have $Deg_{\mathcal{L}}(v') > d(G)$. □

Algorithm Analysis. The space, CPU, and I/O complexities of Algorithm 10 are $O(n), O(m)$ and $O(\frac{l \cdot (m+n)}{B})$ respectively, where l is the number of levels.

Algorithm Design for Edge Deletion. We consider removing an edge. Based on Lemma 6.1, after removing an edge (u, v), if $d(G)$ is unchanged, we only need to update $Deg_{\mathcal{L}}(u)$ and $Deg_{\mathcal{L}}(v)$ according to Eq. 7.

Algorithm Design for Edge Insertion. We consider inserting an edge (u, v). We first update $Deg_{\mathcal{L}}(u)$ and $Deg_{\mathcal{L}}(v)$ based on Eq. 7. According to Lemma 6.1, we need to find a new $level\text{-}value$ for any node v' in G if $Deg_{\mathcal{L}}(v') > d(G)$. Note that the raise of $level\text{-}value$ of a node v' will increase the $upper\text{-}degree$ of its neighbors. This may lead to some nodes $u' \in \mathsf{nbr}(v')$ such that $Deg_{\mathcal{L}}(u') > d(G)$. To minimize such influence, we need to find a minimum new $level\text{-}value$ for node v'. The pseudocode for finding a minimum new $level\text{-}value$ for a node v' is given in Procedure LocalLevel() of Algorithm 11. ml is the maximum $level\text{-}value$ in the graph (line 17). Given a node v', we use an array num to record the number of each $level\text{-}value$ for its neighbors (line 18-19). We initialize the new $level\text{-}value$ $\mathcal{L}_{new}$ for node v' to be $ml + 1$ and use cd to calculate the $upper\text{-}degree$ for the level $\mathcal{L}_{new}$. We find the minimum $\mathcal{L}_{new}$ satisfying $cd \leq d(G)$ (line 21-24) and return $\mathcal{L}_{new}$ as the new $\mathcal{L}(v')$.

The overall algorithm for edge insertion is given in Algorithm 11. SemiInsert* is firstly invoked to check whether $d(G)$ changes. The $Deg_{\mathcal{L}}$ of u and v are updated according to Eq. 7 (line 3-5). Then we adopt Lemma 6.1 and focus on the node v' with $Deg_{\mathcal{L}}(v') > d(G)$ in line 9. In line 13, we invoke LocalLevel (line 16-25) to calculate the new $level\text{-}value$ $\mathcal{L}(v')$. With new $\mathcal{L}(v')$, we invoke ComputeUDeg (line 26-30) in line 14 to calculate the corresponding $Deg_{\mathcal{L}}(v')$ according to Eq. 7 (line 29). In line 15, we invoke UpdateNbrUDeg (line 31-33) to update $Deg_{\mathcal{L}}$ of the neighbors of v'. Based on Eq. 7, only those neighbors u' with $\mathcal{L}(u')$ falling in the range $(\mathcal{L}_{old}, \mathcal{L}(v')]$ will have $Deg_{\mathcal{L}}(u')$ increase by 1 (line 33).

Algorithm 11 DInsert$^*(G, \mathcal{L}, Deg_{\mathcal{L}}, Edge(u, v))$

1: invoke SemiInsert$^*(G, (u, v))$ to update core number;
2: **if** $d(G)$ changes (based on Eq. 5) **then** recompute the index and **return**;
3: **if** $\mathcal{L}(u) \leq \mathcal{L}(v)$ **then** $Deg_{\mathcal{L}}(u) \leftarrow Deg_{\mathcal{L}}(u) + 1$;
4: **if** $\mathcal{L}(v) \leq \mathcal{L}(u)$ **then** $Deg_{\mathcal{L}}(v) \leftarrow Deg_{\mathcal{L}}(v) + 1$;
5: **if** $Deg_{\mathcal{L}}(u) \leq d(G)$ and $Deg_{\mathcal{L}}(v) \leq d(G)$ **then return**;
6: update $\leftarrow$ **true**;
7: **while** update **do**
8: update $\leftarrow$ **false**;
9: **for** $v' \leftarrow v_1$ **to** v_n s.t. $Deg_{\mathcal{L}}(v') > d(G)$ **do**
10: update $\leftarrow$ **true**;
11: load $\mathsf{nbr}(v')$ from disk;
12: $\mathcal{L}_{old} \leftarrow \mathcal{L}(v')$;
13: $\mathcal{L}(v') \leftarrow$ LocalLevel($\mathsf{nbr}(v')$);
14: $Deg_{\mathcal{L}}(v') \leftarrow$ ComputeUDeg($\mathsf{nbr}(v')$);
15: UpdateNbrUDeg($\mathsf{nbr}(v'), \mathcal{L}_{old}, \mathcal{L}(v')$);
16: **Procedure** LocalLevel($\mathsf{nbr}(v')$)
17: $ml \leftarrow max_{u' \in V} \mathcal{L}(u')$
18: $\mathsf{num}(i) \leftarrow 0$ for all $1 \leq i \leq ml$;
19: **for all** $u' \in \mathsf{nbr}(v')$ **do** $\mathsf{num}(\mathcal{L}(u')) \leftarrow \mathsf{num}(\mathcal{L}(u')) + 1$;
20: $cd \leftarrow 0$; $\mathcal{L}_{new} \leftarrow ml + 1$;
21: **for** $i \leftarrow ml$ **downto** 1 **do**
22: $cd \leftarrow cd + \mathsf{num}(i)$;
23: **if** $cd > d(G)$ **then break**;
24: $\mathcal{L}_{new} \leftarrow i$;
25: **return** $\mathcal{L}_{new}$;
26: **Procedure** ComputeUDeg($\mathsf{nbr}(v')$)
27: $ud \leftarrow 0$;
28: **for all** $u' \in \mathsf{nbr}(v')$ **do**
29: **if** $\mathcal{L}(u') \geq \mathcal{L}(v')$ **then** $ud \leftarrow ud + 1$;
30: **return** ud;
31: **Procedure** UpdateNbrUDeg($\mathsf{nbr}(v'), \mathcal{L}_{old}, \mathcal{L}(v')$)
32: **for all** $u' \in \mathsf{nbr}(v')$ **do**
33: **if** $\mathcal{L}_{old} < \mathcal{L}(u') \leq \mathcal{L}(v')$ **then** $Deg_{\mathcal{L}}(u') \leftarrow Deg_{\mathcal{L}}(u') + 1$;

TABLE 6: ILLUSTRATION OF DInsert*(Insert (v_0, v_6))

Iteration\v	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
Init	1	1	2	2	1	2	1	1	1
Iteration 1	2	1	2	2	1	2	2	1	1
Iteration 2	2	1	2	3	1	2	2	1	1

Example 6.3: Table 6 gives an example that we insert an edge (v_0, v_6) into G of Fig. 1. The initial $\mathcal{L}$ of each node is given in the second row. We know the degeneracy $d(G)$ of G is 3. After inserting the edge (v_0, v_6), we have $Deg_{\mathcal{L}}(v_0) = 4$ and $Deg_{\mathcal{L}}(v_0) > d(G)$. Thus we update $\mathcal{L}(v_0)$ from 1 to 2. Similarly, we update $\mathcal{L}(v_6)$ from 1 to 2. After the first iteration, we only have $Deg_{\mathcal{L}}(v_3) = 4 > d(G)$. We update $\mathcal{L}(v_3)$ to 3 in the second iteration. The algorithm terminates after the second iteration. □

Algorithm Analysis. Let l be the number of iterations in Algorithm 11. The space, I/O, and time complexities of Algorithm 11 are $O(n)$, $O(\frac{l \cdot (m+n)}{B})$, and $O(l \cdot (m + n))$ respectively. Here, l is very small in practice. For example, in our experiments, l is not larger than 10 in all the tested datasets.

7 PERFORMANCE STUDIES

In this section, we experimentally evaluate the performance of our proposed algorithms for both k-core and degeneracy order.

TABLE 7: DATASETS (1K = 10^3)

Datasets	$\|V\|$	$\|E\|$	density	k_{max}	CSR size
DBLP	317K	1,049K	3.31	113	9.7 MB
Youtube	1,134K	2,987K	2.63	51	28.4 MB
WIKI	2,394K	5,021K	2.10	131	49.7 MB
CPT	3,774K	16,518K	4.38	64	147.2 MB
LJ	3,997K	34,681K	8.67	360	293.4 MB
Orkut	3,072K	117,185K	38.14	253	949.8 MB
IT	41,291K	1,150,725K	27.86	3224	9.4 GB
Twitter	41,652K	1,468,365K	35.25	2488	11.9 GB
Webbase	118,142K	1,019,903K	8.63	1506	8.6 GB
SK	50,636K	1,949,412K	38.49	4510	15.8 GB
UK	105,896K	3,738,733K	35.30	5704	30.3 GB
Clueweb	978,408K	42,574,107K	43.51	4244	344.5 GB

Subsection 7.1 compares our solutions with state-of-the-art algorithms; Subsection 7.2 shows the efficiency of our maintenance algorithm; We report the scalability of these two algorithms in Subsection 7.5; Subsection 7.3 reveals the efficiency of degeneracy ordering algorithm and Subsection 7.4 focuses on the degeneracy order maintenance.

All algorithms are implemented in C++, using gcc complier at -O3 optimization level. All the experiments are performed under a Linux operating system running on a machine with an Intel Xeon 3.4GHz CPU, 16GB RAM and 7200 RPM SATA Hard Drives (2TB). The cost of each algorithm is measured as the amount of wall-clock time elapsed during the program's execution. We adhere to standard external memory model for I/O statistics [35].

Datasets. We use two groups of datasets to demonstrate the efficiency of our semi-external algorithm. Group one consists of six graphs with relatively small size: DBLP, Youtube, WIKI, CPT, LJ and Orkut. Group two consists of six big graphs: Webbase, IT, Twitter, SK, UK and Clueweb. The detailed information for the 12 datasets is displayed in Table 7. Here, the CSR size is the size of the graph represented using the binary Compressed Sparse Row (CSR) format.

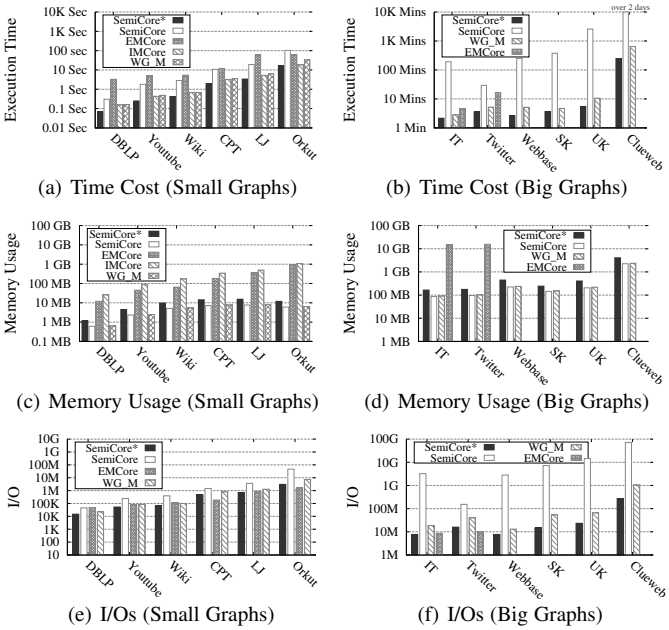
In group one (small graphs), DBLP is a co-authorship network of DBLP. Youtube is a social network based on the user friendship in Youtube. WIKI is a network containing all the users and discussion from the inception of Wikipedia till January 2008. CPT is a citation graph that consists in all citations made by patents granted between 1975 and 1999. LJ (LiveJournal) is a free online blogging community. Orkut is a free online social network.

In group two (big graphs), IT is a fairly large crawl of the .it domain. Twitter is a social network collected from Twitter where nodes are users and edges follow tweet transmission. Webbase is a graph obtained from the 2001 crawl performed by the WebBase crawler. SK is a graph obtained from a 2005 crawl of the .sk domain. UK is a graph gathering a snapshot of about 100 million pages for the DELIS project in May 2007. Finally, Clueweb is a web graph underlying the ClueWeb12 dataset. All datasets can be downloaded from SNAP (http://snap.stanford.edu/) and LAW (http://law.di.unimi.it/).

7.1 Core Decomposition

To explicitly reveal the performance of our core decomposition algorithms, we select the core decomposition algorithm EMCore [1] and the in-memory algorithm [6], denoted by IMCore as comparisons. We also report the results of algorithm WG_M [37], which is a parallel work for core decomposition in large graphs. WG_M also follows the semi-external model. More details of WG_M are given in Section 8.

Small Graphs. As shown in Fig. 5 (a), the total running time of SemiCore* is the fastest. It is 10 times faster than that of the EMCore on average. It is remarkable that SemiCore* can be even faster than the in-memory algorithm IMCore. Fig. 5 (c) shows



(a) Time Cost (Small Graphs) (b) Time Cost (Big Graphs)

(c) Memory Usage (Small Graphs) (d) Memory Usage (Big Graphs)

(e) I/Os (Small Graphs) (f) I/Os (Big Graphs)

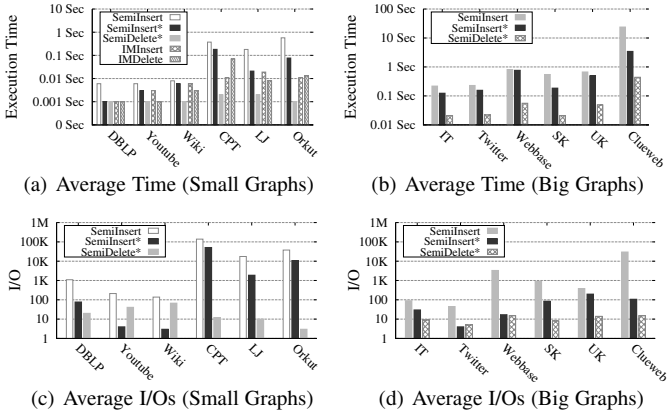
Fig. 5: CORE DECOMPOSITION ON DIFFERENT DATASETS

that algorithm SemiCore* requires less memory than EMCore and IMCore. Among all algorithms, SemiCore uses the least amount of memory since it does not rely on the cnt numbers for all nodes. By contrast, EMCore uses a large amount of memory. Especially in Orkut and CPT, EMCore uses almost the same memory size as IMCore. Fig. 5 (e) shows the I/O consumption of all algorithms except IMCore. SemiCore* and EMCore usually use the least amount of I/Os. However, due to the simple read-only data access of SemiCore*, SemiCore* is much more efficient than EMCore (refer to Fig. 5 (a)).

Big Graphs. We report the performance of our algorithms on big graphs in Fig. 5 (b), (d), and (f). Note that algorithm EMCore can only successfully process IT and Twitter using the machine with 16GB RAM. The largest dataset Clueweb contains nearly 1 billion nodes and 42.6 billion edges. We can see from Fig. 5 (a) that SemiCore* can process all datasets within 10 minutes except Clueweb. In Fig. 5, we can see that SemiCore* totally costs less than 4.2 GB memory to process the largest dataset Clueweb. This result demonstrates that our algorithm can be deploy in any commercial machine to process big graph data. Fig. 5 (f) further reveals the advance of optimization in terms of I/O cost, since SemiCore* spends much less I/Os than SemiCore and WG_M in all datasets.

7.2 Core Maintenance

We test the performance of our maintenance algorithms SemiInsert, SemiInsert*, and SemiDelete*. Since there are not previous algorithms for core maintenance in big graphs, we add the state-of-the-art streaming in-memory algorithms [2] for comparison, which are denoted by IMInsert and IMDelete. In the machine with 16GB RAM, IMInsert and IMDelete can only process the graph updates for the small graphs. We randomly select 100 distinct existing edges in the graph for each test. To test the performance of edge deletion, we remove the 100 edges from the graph one by one and take the average processing time and I/Os. To test the performance of edge insertion, after the 100 edges are removed, we insert them into the graph one by one and take the average processing time and I/Os. The experimental results are reported in Fig. 6.



(a) Average Time (Small Graphs) (b) Average Time (Big Graphs)

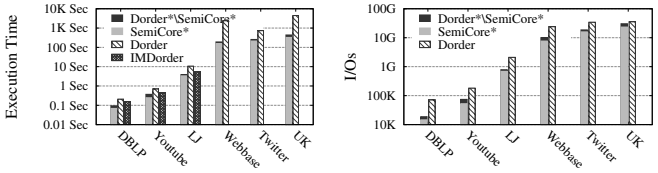
(c) Average I/Os (Small Graphs) (d) Average I/Os (Big Graphs)

Fig. 6: CORE MAINTENANCE ON DIFFERENT DATASETS

From Fig. 6, we can see that SemiDelete* is more efficient than SemiInsert* in both processing time and I/Os for all datasets. This is because SemiDelete* simply follows SemiCore* and does not rely on the calculation of other new graph properties. From Fig. 6 (a), we can find that SemiDelete* is even faster than IMDelete for edge deletion. This is due to the simple structures and data access model used in SemiDelete*. SemiInsert* outperforms SemiInsert in both processing time and I/Os for all datasets.

7.3 Degeneracy Order Computation

We present the performance of the algorithms Dorder and Dorder* for computing the degeneracy order. To explicitly reveal the efficiency of Dorder*, we show the results on three small graphs (DBLP, Youtube and LJ) and three big graphs (Webbase, Twitter and UK). We also report the results of the in-memory algorithm IMDorder [31] on three small graphs for comparison.



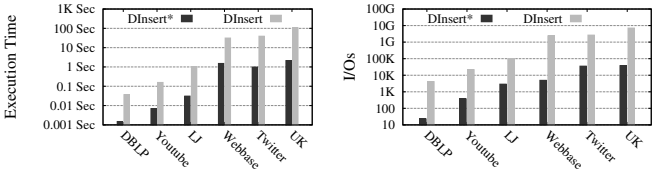
(a) Time Cost (b) I/Os

Fig. 7: TIME COST AND I/OS OF COMPUTING DEGENERACY ORDER

The experimental results are presented in Fig. 7. We show both the time cost and the I/O cost of Dorder* and Dorder. Dorder* contains two phases. The first phase is SemiCore*, which is shown by grey color in the figure. The rest of Dorder* is shown by black color. We can see that after core decomposition, we can only spend small costs to compute the degeneracy order for a graph. In Fig. 7 (a), the Dorder computes the degeneracy order of the three small datasets DBLP, Youtube, and LJ in 0.1 second, 0.3 second, and 3.8 seconds respectively. For the big datasets, Dorder spends 190 seconds, 256 seconds, and 437 seconds for Webbase, Twitter and UK respectively. We can see that with the assistance of SemiCore*, Dorder* is more efficient than Dorder.

7.4 Degeneracy Order Maintenance

We test the performance of our degeneracy maintenance algorithm DInsert* on three small graphs (DBLP, Youtube, and LJ) and three big graphs (Webbase, Twitter and UK). We do not show the performance for edge removal since the removal of an edge will not trigger the update of the $level\text{-}value$ for nodes in the Level-Index. For each test, we select 100 edge insertions that will trigger the change of the Level-Index and take the average cost. We compare our algorithm DInsert* with the Dorder* algorithm



(a) Time Cost (b) I/Os

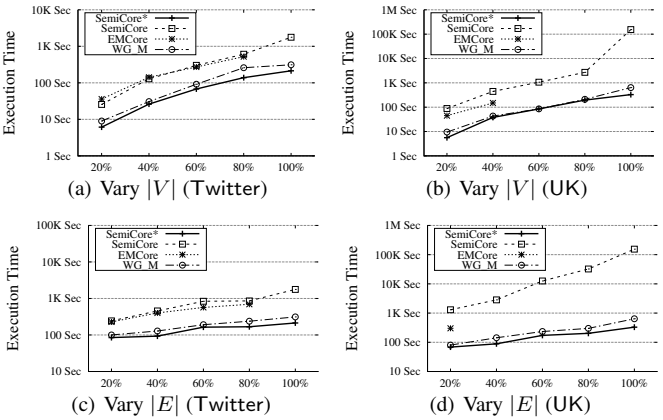
Fig. 8: TIME COST AND I/OS OF DEGENERACY ORDER MAINTENANCE

which works as follows. It first checks whether the graph degeneracy changes using SemiInsert*, if so, it computes the degeneracy order from scratch using Dorder*; otherwise, it invokes line 2-11 of Dorder* (Algorithm 9) to compute the new degeneracy order. Both DInsert* and DInsert include the algorithm to maintain the core number.

The statistics of degeneracy order maintenance are given in Fig. 8. According to Fig. 8, on average, DInsert* is 30 times faster than DInsert and spends 100 times less I/Os comparing to DInsert. As an example, on the Webbase dataset, DInsert* spends 1.5 seconds and $4.9K$ I/Os while DInsert cost 31 seconds and $2387K$ I/Os. This demonstrates the high efficiency of our degeneracy order maintenance algorithm.

7.5 Scalability Testing

In this experiment, we test the scalability of our core decomposition and core maintenance algorithms. We choose two big graphs Twitter and UK for testing. We vary number of nodes $|V|$ and number of edges $|E|$ of Twitter and UK by randomly sampling nodes and edges respectively from 20% to 100%. When sampling nodes, we keep the induced subgraph of the nodes, and when sampling edges, we keep the incident nodes of the edges. Here, we only report the processing time. The memory usage is linear to the number of nodes, and the curves for I/O cost are similar to that of processing time.

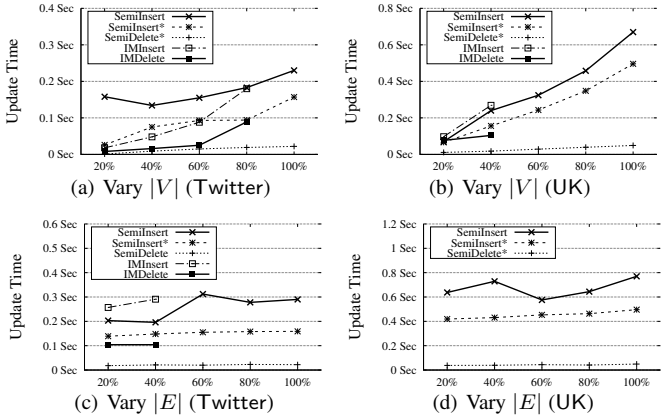


(a) Vary $|V|$ (Twitter) (b) Vary $|V|$ (UK)

(c) Vary $|E|$ (Twitter) (d) Vary $|E|$ (UK)

Fig. 9: SCALABILITY OF CORE DECOMPOSITION

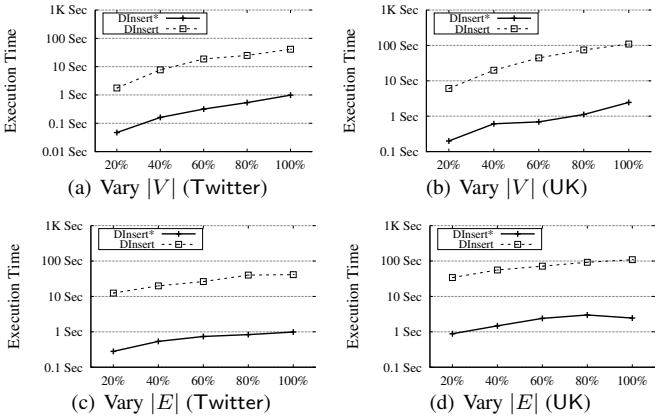
Core Decomposition. Fig. 9 (a) and (b) report the processing time of our proposed algorithms for core decomposition when varying $|V|$ in Twitter and UK respectively. When $|V|$ increases, the processing time for all algorithms increases. SemiCore* performs best in all cases and is over an order of magnitude faster than SemiCore in both Twitter and UK. WG_M is the second best. Fig. 9 (d) and (d) show the processing time of our core decomposition algorithms when varying $|E|$ in Twitter and UK respectively. When $|E|$ increases, the processing time for all algorithms increases, and SemiCore* performs best among all three algorithms. When $|E|$ increases, the gap between SemiCore* and SemiCore also increases. For example, in UK, when $|E|$ reaches 100%, SemiCore* is more than two orders of magnitude faster than SemiCore.



(a) Vary $|V|$ (Twitter) (b) Vary $|V|$ (UK)

(c) Vary $|E|$ (Twitter) (d) Vary $|E|$ (UK)

Fig. 10: SCALABILITY OF CORE MAINTENANCE

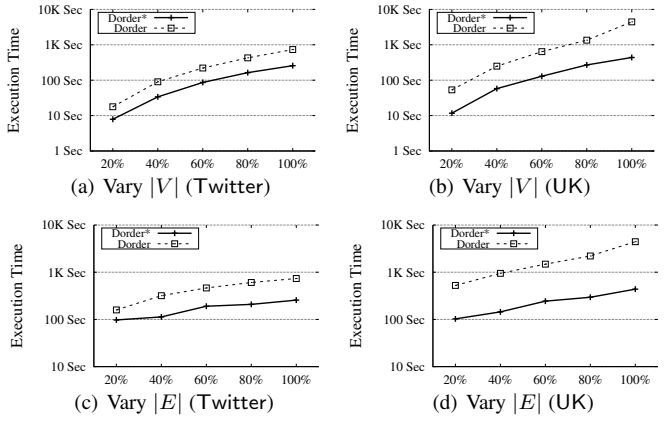


(a) Vary $|V|$ (Twitter) (b) Vary $|V|$ (UK)

(c) Vary $|E|$ (Twitter) (d) Vary $|E|$ (UK)

Fig. 12: SCALABILITY OF DEGENERACY ORDER MAINTENANCE

Core Maintenance. The scalability testing results for core maintenance are shown in Fig. 10. Since there does not exist other work for core maintenance in big graphs, we give the curves for IMInsert and IMDelete as comparisons. As shown in Fig. 10 (a) and Fig. 10 (b), when increasing $|V|$ from 20% to 100%, the processing time for all algorithms increases. SemiDelete* performs best, and SemiInsert* is faster than SemiInsert for all testing cases. The curves of our core maintenance algorithms when varying $|E|$ are shown in Fig. 10 (c) and Fig. 10 (d) for Twitter and UK respectively. SemiDelete* and SemiInsert* are very stable when increasing $|E|$ in both Twitter and UK, which shows the high scalability of our core maintenance algorithms. SemiInsert performs worst among all three algorithms. When $|E|$ increases, the performance of SemiInsert is unstable because SemiInsert needs to locate a connected component whose size can be very large in some cases.



(a) Vary $|V|$ (Twitter) (b) Vary $|V|$ (UK)

(c) Vary $|E|$ (Twitter) (d) Vary $|E|$ (UK)

Fig. 11: SCALABILITY OF DEGENERACY ORDERING

Degeneracy Order Computation. Fig. 11 (a) and (b) report the processing time of our proposed algorithms for degeneracy ordering when varying $|V|$ in Twitter and UK respectively. Fig. 11 (c) and (d) show the statistics when varying $|E|$. When $|V|$ or $|E|$ increases, the processing time for all algorithms increases. Dorder* performs better than Dorder in all cases in both Twitter and UK. Especially, the gap between Dorder* and Dorder increases when $|E|$ increases. When $|E|$ reaches 100% in UK, Dorder* is almost ten times faster than Dorder.

Degeneracy Order Maintenance. Fig. 12 (a) and (b) report the processing time of our proposed algorithms for degeneracy order maintenance when varying $|V|$ in Twitter and UK respectively. Fig. 12 (c) and (d) show the statistics when varying $|E|$. When $|V|$ or $|E|$ increases, the processing time for all algorithms increases. DInsert* performs better than DInsert in all cases in both Twitter and UK.

8 RELATED WORK

k-core is first introduced as a tool for complex network analysis in [5]. Batagelj and Zaversnik [6] give an linear in-memory algorithm for core decomposition, which is presented in Section 3. [1] proposes an I/O efficient algorithm for core decomposition. [36] gives a distributed algorithm for core decomposition. [2] and [3] propose in-memory algorithms to maintain the core numbers of nodes in dynamic graphs. A parallel work of core decomposition for big graphs named WG_M is independently studied by [37]. WG_M also follows the semi-external model and processes each node sequentially. After computing the core number of each node v, WG_M uses a global array to save all neighbors $u \in \mathsf{nbr}(v)$ once the core number of u is not less than that of v, i.e., $\mathsf{core}(u) \geq \mathsf{core}(v)$. In each iteration, WG_M only processes the nodes in the array. Unlike our algorithm, WG_M cannot guarantee that the core numbers of nodes marked in the array will necessarily update.

k-core is also studied in many specific graph structures, such as weighted graphs [7, 38], directed graphs [39], random graphs [16–19] and uncertain graphs [20], etc. Locally computing and estimating core numbers is studied in [40].

The query based community search problem based on k-core model is studied in [25] and [4], where [4] proposes a local search method. [26] gives an influential community model based on k-core. The k-core algorithm is also used as a subroutine to solve other problems, such as finding approximation of densest subgraph and computing cliques with size k [41].

Closely related to core decomposition, graph degeneracy is first defined in [42]. A greedy algorithm for computing the graph degeneracy order is proposed in [31]. Graph degeneracy can be applied in graph clustering [32, 33] and graph coloring [31]. Recently, [30] gives an approximate algorithm for computing the degeneracy order in large graphs.

9 CONCLUSIONS

In this paper, considering that many real-world graphs are big and cannot reside in the main memory of a machine, we study I/O efficient core decomposition on web-scale graphs, which has a large number of applications. The existing solution does not scale to big graphs because it may load most of the graph in memory. Therefore, we follow a semi-external model, which can bound the memory size well. We propose an I/O efficient semi-external algorithm for core decomposition, and explore two optimization

strategies to further reduce the I/O and CPU time. We further propose semi-external algorithms and optimization techniques to handle graph updates. We show that our core decomposition and core maintenance algorithms can be respectively used to efficiently compute the degeneracy order in a graph and maintain the order incrementally when the graph dynamically updates. We conduct extensive experiments on 12 real graphs, one of which contains 978.5 million nodes and 42.6 billion edges, to demonstrate the efficiency of our proposed algorithms.

REFERENCES

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

[2] A. E. Saríyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and U. V. Çatalyürek, "Streaming algorithms for k-core decomposition," *PVLDB*, vol. 6, no. 6, 2013.

[3] R. Li, J. X. Yu, and R. Mao, "Efficient core maintenance in large dynamic graphs," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 10, 2014.

[4] W. Cui, Y. Xiao, H. Wang, and W. Wang, "Local search of communities in large graphs," in *Proc. of SIGMOD'14*, 2014.

[5] S. B. Seidman, "Network structure and minimum degree," *Social networks*, vol. 5, no. 3, 1983.

[6] V. Batagelj and M. Zaversnik, "An o(m) algorithm for cores decomposition of networks," *CoRR*, vol. cs.DS/0310049, 2003.

[7] C. Giatsidis, D. M. Thilikos, and M. Vazirgiannis, "Evaluating cooperation in communities with the k-core structure," in *Proc. of ASONAM'11*, 2011.

[8] A. Verma and S. Butenko, "Network clustering via clique relaxations: A community based approach," *Graph Partitioning and Graph Clustering*, vol. 588, 2012.

[9] 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*, vol. 1, 2006.

[10] 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'05*, 2005.

[11] J. I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani, "k-core decomposition: a tool for the visualization of large scale networks," *CoRR*, vol. abs/cs/0504107, 2005.

[12] M. Altaf-Ul-Amine, K. Nishikata, T. Korna, T. Miyasato, Y. Shinbo, M. Arifuzzaman, C. Wada, M. Maeda, T. Oshima, H. Mori, and S. Kanaya, "Prediction of protein functions based on k-cores of protein-protein interaction networks and amino acid sequences," *Genome Informatics*, vol. 14, 2003.

[13] G. Bader and C. Hogue, "An automated method for finding molecular complexes in large protein interaction networks," *BMC Bioinformatics*, vol. 4, no. 1, 2003.

[14] 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*, vol. 53, no. 2, 2010.

[15] S. N. Dorogovtsev, A. V. Goltsev, and J. F. F. Mendes, "K-core organization of complex networks," *Physical review letters*, vol. 96, no. 4, 2006.

[16] T. Luczak, "Size and connectivity of the k-core of a random graph," *Discrete Math.*, vol. 91, no. 1, 1991.

[17] B. Pittel, J. Spencer, and N. Wormald, "Sudden emergence of a giant k-core in a random graph," *J. Comb. Theory Ser. B*, vol. 67, no. 1, 1996.

[18] M. Molloy, "Cores in random hypergraphs and boolean formulas," *Random Struct. Algorithms*, vol. 27, no. 1, 2005.

[19] S. Janson and M. J. Luczak, "A simple solution to the k-core problem," *Random Struct. Algorithms*, vol. 30, no. 1-2, 2007.

[20] F. Bonchi, F. Gullo, A. Kaltenbrunner, and Y. Volkovich, "Core decomposition of uncertain graphs," in *Proc. of KDD'14*, 2014.

[21] B. Balasundaram, S. Butenko, and I. V. Hicks, "Clique relaxations in social network analysis: The maximum k-plex problem," *Operations Research*, vol. 59, no. 1, 2011.

[22] R. Andersen and K. Chellapilla, "Finding dense subgraphs with size bounds," in *Algorithms and Models for the Web-Graph*, 2009.

[23] L. Qin, R. Li, L. Chang, and C. Zhang, "Locally densest subgraph discovery," in *Proc. of KDD'15*, 2015.

[24] J. Healy, J. Janssen, E. Milios, and W. Aiello, "Characterization of graphs using degree cores," in *Algorithms and Models for the Web-Graph*, 2008.

[25] M. Sozio and A. Gionis, "The community-search problem and how to plan a successful cocktail party," in *Proc. of KDD'10*, 2010.

[26] R. Li, L. Qin, J. X. Yu, and R. Mao, "Influential community search in large networks," *PVLDB*, vol. 8, no. 5, 2015.

[27] Z. Zhang, J. X. Yu, L. Qin, L. Chang, and X. Lin, "I/O efficient: computing sccs in massive graphs," in *Proc. of SIGMOD'13*, 2013.

[28] Z. Zhang, J. X. Yu, L. Qin, and Z. Shang, "Divide & conquer: I/O efficient depth-first search," in *Proc. of SIGMOD'15*, 2015.

[29] Y. Liu, J. Lu, H. Yang, X. Xiao, and Z. Wei, "Towards maximum independent sets on massive graphs," *PVLDB*, vol. 8, no. 13, 2015.

[30] M. Farach-Colton and M.-T. Tsai, "Computing the degeneracy of large graphs," in *Latin American Symposium on Theoretical Informatics*, 2014.

[31] D. W. Matula and L. L. Beck, "Smallest-last ordering and clustering and graph coloring algorithms," *J. ACM*, vol. 30, no. 3, 1983.

[32] C. Giatsidis, F. D. Malliaros, D. M. Thilikos, and M. Vazirgiannis, "Corecluster: A degeneracy based graph clustering framework," in *Proc. of AAAI*, 2014.

[33] M. Charikar, "Greedy approximation algorithms for finding dense components in a graph," in *Proc. of the Third International Workshop on Approximation Algorithms for Combinatorial Optimization*, 2000.

[34] 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.

[35] A. Aggarwal and S. Vitter, Jeffrey, "The input/output complexity of sorting and related problems," *Commun. ACM*, vol. 31, no. 9, 1988.

[36] A. Montresor, F. De Pellegrini, and D. Miorandi, "Distributed k-core decomposition," *TPDS*, vol. 24, no. 2, 2013.

[37] W. Khaouid, M. Barsky, V. Srinivasan, and A. Thomo, "K-core decomposition of large networks on a single pc," *PVLDB*, vol. 9, no. 1, 2015.

[38] M. Eidsaa and E. Almaas, "S-core network decomposition: A generalization of k-core analysis to weighted networks," *Physical Review E*, vol. 88, 2013.

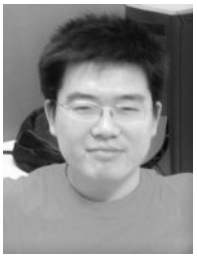
[39] C. Giatsidis, D. M. Thilikos, and M. Vazirgiannis, "D-cores: Measuring collaboration of directed graphs based on degeneracy," in *Proc. of ICDM'11*, 2011.

[40] M. P. O'Brien and B. D. Sullivan, "Locally estimating core numbers," in *Proc. of ICDM'14*, 2014.

[41] V. E. Lee, N. Ruan, R. Jin, and C. Aggarwal, "A survey of algorithms for dense subgraph discovery," in *Managing and Mining Graph Data*, 2010.

[42] D. R. Lick and A. T. White, "k-degenerate graphs," *CJM*, vol. 22, 1970.

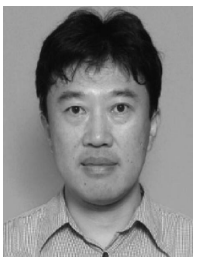
Dong Wen reveived the BEng degree in the School of Software from Nankai University in 2013. He is currently working toward the PhD degree in the Centre of Quantum Computation and Intelligent Systems, University of Technology, Sydney. His major research interests include community detection and I/O efficient algorithms on massive graphs.



Lu Qin received the BEng degree from Renmin University of China in 2006, and the PhD degree from the Department of Systems Engineering and Engineering Management, Chinese University of Hong Kong in 2010. He is currently a postdoc research fellow in the Centre of Quantum Computation and Intelligent Systems, University of Technology, Sydney. His research interests include parallel big graph processing, I/O efficient algorithms on massive graphs, and keyword search in relational database.

Ying Zhang received the Bachelor and Master degree in Computer Science from Peking University, P.R. China, in 1998 and 2001. He got the PhD from the School of Computer Science and Engineering, the University of New South Wales (UNSW) in 2008. He is currently a senior lecture in the Centre of Quantum Computation and Intelligent Systems, University of Technology, Sydney. His research interests include efficient query processing on spatial data, stream data and graphs.

Xuemin Lin received the BSc degree in applied math from Fudan University in 1984, and the PhD degree in computer science from the University of Queensland in 1992. He is a professor in the School of Computer Science and Engineering, University of New South Wales. His current research interests lie in approximate query processing, spatial data analysis, and graph visualization. He is currently an associate editor of the ACM Transactions on Database Systems, a fellow member of the IEEE.



Jeffrey Xu Yu received the BE, ME, and the PhD degrees in computer science, from the University of Tsukuba, Japan, in 1985, 1987, and 1990, respectively. He is currently a professor at the Department of Systems Engineering and Engineering Management, The Chinese University of Hong Kong. His major research interests include graph mining, graph database, keyword search, and query processing and optimization. He is a senior member of the IEEE, a member of the IEEE Computer Society, and a member of ACM.