I/O-Efficient Algorithms for Degeneracy Computation on Massive Networks

Rong-Hua Li, Qiushuo Song, Xiaokui Xiao, Lu Qin, Guoren Wang, Jeffrey Xu Yu, and Rui Mao

Abstract—Degeneracy is an important concept to measure the sparsity of a graph which has been widely used in many network analysis applications. Many network analysis algorithms, such as clique enumeration and truss decomposition, perform very well in graphs having small degeneracies. In this paper, we propose an I/O-efficient algorithm to compute the degeneracy of the massive graph that cannot be fully kept in the main memory. The proposed algorithm only uses $O(n)$ memory, where n denotes the number of nodes of the graph. We also develop an I/O-efficient algorithm to incrementally maintain the degeneracy on dynamic graphs. Extensive experiments show that our algorithms significantly outperform the state-of-the-art degeneracy computation algorithms in terms of both running time and I/O costs. The results also demonstrate high scalability of the proposed algorithms. For example, in a real-world web graph with 930 million nodes and 13.3 billion edges, the proposed algorithm takes only 633 seconds and uses less than 4.5GB memory to compute the degeneracy.

Index Terms—Degeneracy, I/O-efficient algorithm, k-core, massive graphs

1 Introduction

Given a graph G, the degeneracy of G, denoted by δ, is the smallest integer such that every subgraph of G has a node of degree at most δ. The degeneracy has been recognized as an important concept for measuring the sparsity of a graph, and it finds applications in several different domains, including network analysis, graph mining, and graph theory. A few significant applications are as follows.

Maximal Clique Enumeration. A clique is a completed subgraph in which every pair of nodes has an edge, and a *maximal* clique is one whose super-graphs are all non-cliques. The state-of-the-art algorithms [1], [2] for enumerating maximal cliques require an efficient algorithm for deriving the *degeneracy ordering* of nodes, which is a byproduct of degeneracy computation. Therefore, an improved algorithm for computing degeneracy immediately leads to more efficient methods for maximal cliques enumeration.

Densest Subgraph Discovery. The *densest subgraph* G' [3] of a graph is the one that maximizes m'/n', where m' and n' denote the numbers of edges and nodes in G'. The identification of the densest subgraph has numerous applications such as community discovery [4], [5], [6], [7], [8], graph compression [9], computational biology [10], and spam detection [11]. Since the exact computation of densest subgraph is expensive [3], most existing techniques aim to derive approximate solutions, which require obtaining an approximation of the maximum subgraph density, i.e., the maximum value of m'/n'. It is well known that the degeneracy is a 2-approximation of the maximum subgraph density [12], and therefore an efficient algorithm for computing degeneracy is highly useful for densest subgraph computation [12], [13], [14].

Complexity Bounds of Graph Algorithms. Degeneracy is a 2-approximation of *arboricity* [15], [16] (see Section 2 for details). The arboricity is a classic graph measure that is frequently used to analyze the space or time complexity of network analysis algorithms, such as triangle counting [17], k-clique enumeration [18], truss decomposition [19], [20], structural graph clustering [21], influential community search [22], [23], top-k structural diversity search [24]. Computing the exact value of arboricity, however, incurs significant costs [25]. To address this issue, one can derive the degeneracy of the input graph G, and then use it as an approximation of G's arboricity for analysis.

In addition, the degeneracy δ has also been widely used as a parameter in many fixed-parameter tractable (FPT) graph algorithms [26], in which the complexity of these algorithms depend mainly on an exponential function of δ, e.g., $O(3^{\delta})$. For example, the classic dominating set problem [27], [28], [29], cycle counting problem [30], as well as the maximal clique enumeration problem are shown to be FPT with the parameter δ. Thus, computing the degeneracy of a graph G can be useful to predict whether such FPT algorithms are tractable in G.

- *Rong-Hua Li is with the Beijing Institute of Technology, Beijing 100811, China, and also with the National Engineering Laboratory for Big Data System Computing Technology, Beijing, China. E-mail: lironghuabit@126.com.*
- *Qiushuo Song and Rui Mao are with the Shenzhen Institute of Computing Sciences, Guangdong Province Key Laboratory of Popular High Performance Computers, Shenzhen University, Shenzhen, Guangdong 518060, China. E-mail: qsong98@gmail.com, mao@szu.edu.cn.*
- *Xiaokui Xiao is with the National University of Singapore, Singapore 119077. E-mail: xkxiao@ntu.edu.sg.*
- *Lu Qin is with the University of Technology, Sydney, NSW 2007, Australia. E-mail: Lu.Qin@uts.edu.au.*
- *Guoren Wang is with the Beijing Institute of Technology, Beijing 100811, China. E-mail: wanggrbit@126.com.*
- *Jeffrey Xu Yu is with the Chinese University of Hong Kong, Hong Kong. E-mail: yu@se.cuhk.edu.hk.*

Manuscript received 2 Feb. 2018; revised 10 Aug. 2020; accepted 29 Aug. 2020. Date of publication 3 Sept. 2020; date of current version 3 June 2022. (Corresponding author: Guoren Wang.) Recommended for acceptance by Y. Xia. Digital Object Identifier no. 10.1109/TKDE.2020.3021484

Motivation. For a graph G that fits in the main memory, the degeneracy of G can be computed efficiently using a linear-time algorithm for *core decomposition* [31], [32]. Specifically, the algorithm consists of several iterations, such that the kth ($k = 1, 2, \ldots$) iteration recursively removes all nodes in G whose degrees are smaller than k, until all remaining nodes have degrees at least k in the subgraph that they induce (this subgraph is referred to as the *k-core*). It is known that if the degeneracy of G equals δ, then the algorithm runs in exactly δ iterations, i.e., δ equals the largest *core number* k in G.

Nevertheless, real-world graphs are often too large for the main memory of a single machine. For example, the current Facebook social network contains 1.32 billion nodes and 140 billion edges (http://newsroom.fb.com/company-info). This motivates *semi-external* algorithms for degeneracy computation via k-core decomposition [33], which require only the nodes of G to be memory-resident but allows the edges of G to be disk-resident. For instance, for the aforementioned Facebook graph, around 10GB memory is sufficient to accommodate all nodes in the graph.

The state-of-the-art semi-external algorithm for core decomposition [33], however, suffers from the following deficiencies. First, to derive the degeneracy δ of a graph G, it requires enumerating the 1-, 2-, $\ldots$, δ-cores of G, which incurs unnecessary overheads because, intuitively, the i-cores ($1 \leq i \leq \delta - 1$) are not particularly useful for degeneracy computation. Second, if we use this algorithm to track the degeneracy of a dynamic graph G, we would need to maintain the core decomposition of G which takes $O(l(m + n)/B)$ I/O costs [33] (l is the iteration number of the algorithm, m and n denote the number of edges and nodes of the graph respectively, and B denotes the block size), thus it is rather costly for massive graphs. Alternatively, one may apply the existing *semi-streaming*[1] algorithms [12], [13], [34] for degeneracy computation. These algorithms, however, can only return $(2 + \epsilon)$ approximation of degeneracy and are designed only for static graphs (see Section 3 for details).

Our Contributions. To overcome the limitations of the existing solutions, we propose a semi-external method for degeneracy computation that utilizes an algorithm design drastically different from previous methods. Specifically, our method does not rely on core decomposition to identify the degeneracy δ of the input graph G. Instead, we start by deriving an (potentially loose) upper bound ub and a lower bound lb of δ, and then perform a binary search in the range $[lb, ub]$ to pinpoint the exact value of δ. To facilitate this binary search, we develop a novel I/O-efficient algorithm that takes as input G and an integer k, and returns a k-core of G (if any) without computing the full core decomposition. In addition, we also devise a semi-external algorithm to incrementally maintain the degeneracy of G when there are edge insertions or deletions.

We experimentally evaluate our algorithms using a variety of benchmark datasets with up to several billion edges. The results show that our degeneracy computation method is an order of magnitude faster than the state-of-the-art solution [33], and our degeneracy maintenance approach is up to three orders of magnitude faster than prior art. For instance, on the GSH dataset with 0.9 billion nodes and 13.3 billion edges, our algorithm takes around 10 minutes to derive the exact value of degeneracy, whereas the state of the art requires more than two hours. For degeneracy maintenance, our solution needs only 0.02 seconds (resp. 0.1 milliseconds) on average to process an edge insertion (resp. deletion), whereas prior art requires around 0.3 seconds (resp. 0.1 seconds). Furthermore, our solution is memory-efficient: it requires less than 4.5GB memory to handle GSH, which is 625GB in size.

Taking one step further in our experiments, we apply our algorithm to measure the degeneracies of 150 publicly available graphs, including social networks, web graphs, citation networks, collaboration networks, infrastructure networks, biological networks, and communication networks. This large experimental study is motivated by the facts that (i) a large body of existing work (e.g., [2], [17], [18], [22], [28], [29], [35]) assume that real networks have small degeneracies, but (ii) to our knowledge, this assumption has never been validated with systematic experiments, presumably because of the significant overheads incurred by existing algorithms for degeneracy computation. Our results show two sides of a coin. On one hand, we observe that the majority of the 150 graphs tested do have fairly small degeneracies (with $\delta < 200$); on the other hand, we also notice that large social networks and web graphs can have degeneracies up to several thousands. In particular, the degeneracies of a social network Twitter and a web graph UK are 2,488 and 10,424, respectively. This indicates that the "small-degeneracy" assumption might be excessively optimistic for social networks and web graphs, and that future work on these two types of graphs should not rely on this assumption.

Organization. We formally define our problem in Section 2, and survey the existing I/O-efficient algorithms for degeneracy computation in Section 3. Sections 4 elaborates the I/O-efficient degeneracy computation algorithm, and Section 5 describes the I/O-efficient degeneracy maintenance algorithm. Section 6 presents the experimental results. Finally, we conclude this work in Section 7.

2 Preliminaries

Problem definition. We aim to develop efficient algorithms for (i) computing the degeneracy δ of a graph G and (ii) incrementally maintain δ when there are edge insertions or deletions in G. We assume that G is *massive* in the sense that the main memory can only accommodate G's nodes but not its edges. In other words, we assume that the memory size is $O(n)$. Note that this assumption is well-adopted in previous work for analyzing massive graphs [12], [33].

Below, we introduce some useful notations, as well as the formal definition of the degeneracy δ of a graph G.

Concepts and Notations. Let $G = (V, E)$ be an undirected graph with a node set V and an edge set E, with $|V| = n$ and $|E| = m$. Let $N_u(G) \triangleq \{v \,|\, (u, v) \in E\}$ be the set of neighbors of u in G, and $d_u(G) = |N_u(G)|$ denote the degree of u in G. A graph $G' = (V', E')$ is a subgraph of G, denoted as $G' \subseteq G$, if $V' \subseteq V$ and $E' \subseteq E$. Give a set of node $V_s \subseteq V$, the subgraph induced by V_s is defined as $G(V_s) = (V_s, E_s)$, where $E_s = \{(u, v) \,|\, (u, v) \in E, u \in V_s, v \in V_s\}$.

The *degeneracy* of G [36], denoted as δ, is defined below.

1. A semi-streaming algorithm is a semi-external algorithm that requires only a small number of sequential passes of the input graph.

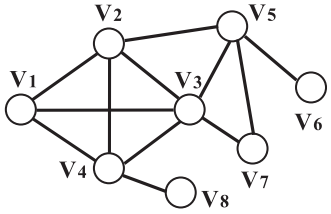


Fig. 1. Running example.

Definition 1 (Degeneracy). *The degeneracy δ of a graph G is the smallest integer such that every nonempty subgraph of G contains a node with degree at most δ. More formally,*

$$\delta \triangleq \max_{\forall G' \subseteq G} \min_{u \in G'} \{d_u(G')\}. \quad (1)$$

Given a graph G and an integer k, the *k-core* of G, denoted as C_k, is the maximal induced subgraph of G such that every node in C_k has degree no less than k [37], i.e., $d_u(C_k) \geq k$ for every $u \in C_k$. The *core number* of a node u, denoted as c_u, is the largest integer k such that there is a k-core containing u. The maximum core number of a graph G, denoted by $c_{\max}$, is the maximum value of core number for any node in G. It is known that the degeneracy of G equals the maximum core number [2], i.e., $\delta = c_{\max}$. In the remainder of the paper, we use δ and $c_{\max}$ interchangeably to denote the degeneracy of G. We demonstrate the above concepts using an example below.

Example 1. Consider the graph G shown in Fig. 1. The degeneracy of G is 3, because (i) there is a subgraph induced by $\{v_1, v_2, v_3, v_4\}$ where the minimum node degree is 3, and (ii) no subgraph has minimum degree larger than 3. In addition, the core number of each node in $\{v_1, v_2, v_3, v_4\}$ is 3, because the subgraph induced by $\{v_1, v_2, v_3, v_4\}$ is a 3-core. Meanwhile, the core numbers of v_5 and v_7 are equal to 2, and the core numbers of v_6 and v_8 equal 1. □

Graph Storage and I/O Model. We organize G on the disk in the same manner as in previous work [33]. Specifically, we store the adjacency lists of G, denoted as $\{N_{v_1}(G), N_{v_2}(G), \ldots, N_{v_n}(G)\}$, in an *edge file* sequentially on the disk. We also use a *node file* to store a list including the offsets and degrees of the nodes $\{v_1, v_2, \ldots v_n\}$. To load the neighbors of a node v_i into the memory, we first access the node file to get the offset and degree of v_i, and then load the neighbors of v_i from the edge file. We adopt the widely-used external memory model proposed in [38] to analyze the I/O-efficient algorithm. Specifically, let M be the memory size and B be the block size ($B < M$). The disk files are organized by blocks and each block size is B bytes. For each read I/O, the algorithm loads one block of size B from disk into main memory. Similarly, for each write I/O, the algorithm write one block of size B from the main memory into disk. The I/O costs for each algorithm denotes the total number of read and write I/Os taken by the algorithm. Note that the semi-external I/O model assumes the memory size $M = O(n)$ [12], [33], i.e., the main memory can hold all nodes of the graph but cannot store all edges. In this paper, we adopt such a semi-external I/O model to design and analyze algorithms for degeneracy computation.

3 Existing I/O-Efficient Algorithms

In the literature, there exist two types of algorithms for degeneracy computation that assumes $O(n)$ memory as we do. The first type is *semi-streaming* algorithms [12], [13], [14], [34] that require only a small number of sequential passes of the input graph, while the second type is a semi-external algorithm for k-core decomposition [33], referred to as SemiCore. In this section, we reviews two types of algorithms in detail.

We also note that there is a *full external-memory* k-core decomposition algorithm [39] designed for the case when the memory is too small to accommodate even the nodes in the input graph. Such a *full external-memory* takes $O(\delta(m+n)/B)$ I/Os. As shown in [33], the performance of this *full external-memory* algorithm is much worse than the state-of-the-art semi-external algorithm [33] which uses $O(l(m+n)/B)$ I/Os (l is typically smaller than δ). Therefore, we omit the *full external-memory* algorithm proposed in [39] in this section.

3.1 Semi-Streaming Algorithms

Existing semi-streaming algorithms [12], [13], [34] adopt a greedy multi-pass approach to compute degeneracy. Specifically, in the ith pass, the algorithms identify an induced subgraph $G_i = (V_i, E_i)$ and compute the *density* ρ_i of G_i, where $\rho_i = |E_i|/|V_i|$. Then, they delete all nodes whose degrees are smaller than $\alpha \times \rho_i$, where $\alpha = 2 + \epsilon > 2$ is a given parameter. When all nodes are removed, the algorithms terminate and output $\alpha \times \max_i\{\rho_i\}$ as an α-approximation of the degeneracy. Throughout the algorithms, we only maintain the degree of each node in the main memory, which takes only $O(n)$ space. It was shown that such semi-streaming algorithms only require $O(\log_{1+\varepsilon/2} n)$ passes over G [12].

The main drawback of the above semi-streaming algorithms is that their approximation ratio is relatively loose, as demonstrated in the experiments. Specifically, the algorithms can only provide $(2+\epsilon)$-approximate solutions when incurring $O(\log_{1+\varepsilon/2} n \times (m+n)/B)$ I/O costs, where B denotes the block size. Additionally, it is not clear how the algorithms can be applied to incrementally maintain the degeneracy when G is updated.

One-Pass Streaming Algorithm. In [14], Farach-Colton and Tsai propose a one-pass streaming algorithm to compute $(1+\epsilon)$-approximations of the degeneracy based on a streaming sampling technique. This algorithm, however, requires $O(\epsilon^{-2} n (\log_2 n)^2)$ bits [14] ($\approx O(\epsilon^{-2} n (\log_2 n))$ bytes), which is often larger than the $O(n)$ memory that we assume, especially when ϵ is small.

3.2 The SemiCore Algorithm

The SemiCore algorithm [33] is the state-of-the-art semi-external algorithm to compute the exact degeneracy of a graph, and it is based on iterative k-core computation [40]. To explain the algorithm, we first introduce h-index [41], which is a key concept in SemiCore.

Definition 2 (h-index). *Let $X = \{x_1, x_2, \ldots, x_t\}$ be a set of real values. The* h-index *of X is defined as the largest integer k such that there are k values in X no less than k, i.e., $h(X) \triangleq \arg\max_k\big(|\{x_i \,|\, x_i \geq k, x_i \in X\}| \geq k\big)$.*

TABLE 1
Comparison of Algorithms ($t = \log_{1+\varepsilon/2} n$, $\gamma = \log_2 h$)

Algorithm	Running time	Memory	I/O Cost
[12], [13], [34]	$O((m+n)t)$	$O(n)$	$O(\frac{(m+n)}{B}t)$
[14]	$O(m+n)$	$O(\frac{n(\log_2 n)}{\varepsilon^2})$	$O(\frac{m+n}{B})$
[33]	$O(l(m+n))$	$O(n)$	$O(l\frac{(m+n)}{B})$
SemiDeg	$O(\tau(m+n)\gamma)$	$O(n)$	$O(\tau\frac{(m+n)}{B}\gamma)$
SemiDeg+	$O(\tau(\tilde{n}+\tilde{m})\gamma)$	$O(n)$	$O(\tau\frac{(\tilde{n}+\tilde{m})}{B}\gamma)$

For example, consider the set X of node degrees in the graph in Fig. 1, i.e., $X = \{d_{v_1}, d_{v_2}, \ldots, d_{v_8}\} = \{3, 4, 5, 4, 4, 1, 2, 1\}$. The h-index of X equals 4, since (i) there are four nodes $\{v_2, v_3, v_4, v_5\}$ with degrees no less than 4, and (ii) 4 is the maximum integer satisfying this degree constraint.

The h-index was originally proposed as a measure of the scientific outputs of researchers, but recently was applied to devise efficient graph algorithms [35], [42], [43]. A crucial observation utilized in SemiCore is that the core number of a node u is equal to the h-index of the core numbers of u's neighbors [40]. Based on this observation, SemiCore starts by setting an upper bound of the core number for each node u (e.g., the degree d_u), and then it iteratively refines the upper bound by computing the h-index of the upper bounds of u's neighbors. The algorithm terminates when no node's upper bound needs to be updated [33], [40]. We note that Lü *et al.* [44] also independently discovered such an h-index iteration algorithm. To reduce the I/O costs, SemiCore leverages a clever pruning rule to avoid refining the upper bound of a node until necessary. As shown in [33], the memory overhead of SemiCore is $O(n)$, and the I/O complexity of SemiCore is $O(l \times (m+n)/B)$, where l denotes the number of iterations. In addition, it is shown that SemiCore can be extended to incrementally maintain the core numbers for all nodes when there are edge insertions or deletions.

The main deficiency of SemiCore is that, if we apply it to compute the degeneracy δ of a graph, then it may require a large number of iterations, as it needs to derive the core numbers of all nodes before obtaining δ, leading to significant overheads. Table 1 summarizes the detailed properties of all the existing I/O-efficient algorithms.

4 Our Solution

In this section, we first propose a basic algorithm (referred to as SemiDeg) based on the idea of binary search, and present an improved methods (referred to as SemiDeg+) that offers higher efficiency.

4.1 The Basic Algorithm

Bounds of the Degeneracy. Before presenting the details of SemiDeg, we first introduce several bounds on the degeneracy δ that SemiDeg utilizes. Let $\hat{c}_u$ denote an upper bound of the core number of a node u, and $\hat{\mathbf{c}} = \{\hat{c}_{v_1}, \ldots, \hat{c}_{v_n}\}$ be a set of upper bounds of the core numbers of $v_1, v_2, \ldots, v_n$. In addition, let $\mathbf{d} = \{d_{v_1}, \ldots, d_{v_n}\}$ the set of degrees of the nodes in V. By Definition 2, the h-index of $\hat{\mathbf{c}}$, denoted by $h(\hat{\mathbf{c}})$, is

$$h(\hat{\mathbf{c}}) = \arg\max_k \big(|\{\hat{c}_v \,|\, \hat{c}_v \geq k, v \in V\}| \geq k\big). \quad (2)$$

Given any upper bounds set $\hat{\mathbf{c}}$ of the core numbers, we can easily derive that $h(\hat{\mathbf{c}}) \geq \delta$. Let $h(\hat{\mathbf{c}}, N_u(G))$ be the h-index of u with respect to (w.r.t.) the upper bounds of the core numbers of u's neighbor nodes. By Definition 2,

$$h(\hat{\mathbf{c}}, N_u(G)) \triangleq \arg\max_k \big(|\{\hat{c}_v \,|\, \hat{c}_v \geq k, v \in N_u(G)\}| \geq k\big). \quad (3)$$

For any node $u \in V$, we can easily show that $h(\hat{\mathbf{c}}, N_u(G)) \geq c_u$ for any upper bounds set $\hat{\mathbf{c}}$. Since $d_u \geq c_u$ for node $u \in V$, we have $h(\mathbf{d}, N_u(G)) \geq c_u$. For convenience, we refer to $h_u = h(\mathbf{d}, N_u(G))$ as the h-index of a node u. Let $\mathbf{h} = \{h_{v_1}, \ldots, h_{v_n}\}$ be the set of h-index of all nodes in V. Since $\mathbf{h}$ is a valid upper bounds set of the core numbers, the h-index of $\mathbf{h}$, denoted as h^*, is an upper bound of the degeneracy δ. In what follows, we show that h^* is a tighter upper bound than $h(\mathbf{d})$.

Lemma 1. $h^* \leq h(\mathbf{d})$.

Proof. Since $h_u \leq d_u$ for any $u \in V$, the h-index over $\mathbf{h}$ must be no larger than the h-index over $\mathbf{d}$. As a result, we have $h^* \leq h(\mathbf{d})$. □

Besides the above upper bounds of δ, we can also use $\lceil \frac{m}{n-1} \rceil$ as a lower bound of the degeneracy δ [35].

Algorithm 1. SemiDeg (G)

Input: $G = (V, E)$ in the disk
Output: The degeneracy δ of G

```
1: Let d be the degree set of all nodes in V;
2: for each u ∈ V do
3:   Load N_u(G) from disk;
4:   h_u ← Hindex(u, d, N_u(G));
5: u_max ← arg max_{u∈V}{h_u}; h ← {h_{v_1}, ..., h_{v_n}};
6: lb ← Σ_{u∈V} d_u/2(n − 1); ub ←Hindex (u_max, h, V);
7: while lb ≤ ub do
8:   mid ← ⌊(lb + ub)/2⌋;
9:   update ← 1;   R ← V;
10:  ĉ_u ← d_u for each u ∈ V;
11:  while update = 1 do
12:    update ← 0;
13:    for u ∈ R s.t. ĉ_u < mid do
14:      R ← R \ {u};   update ← 1;
15:      Load N_u(G) from disk;
16:      for v ∈ N_u(G) ∩ R do
17:        ĉ_v ← ĉ_v − 1;
18:  if R ≠ ∅ then lb ← mid + 1; δ ← mid;
19:  else ub ← mid − 1;
20: return δ;
21: Procedure Hindex (u, d, V_s)
22: b(i) ← 0 for all 1 ≤ i ≤ d_u;
23: for each v ∈ V_s do
24:   i ← min{d_v, d_u};   b(i) ← b(i) + 1;
25: sum ← 0; j ← d_u;
26: while j ≥ 1 do
27:   sum ← sum + b(j);
28:   if sum ≥ j then break;
29:   j ← j − 1;
30: return j;
```

Key Idea of SemiDeg. The rationale of SemiDeg is to apply a binary search in $[\lceil \frac{m}{n-1} \rceil, h^*]$ to identify the precise value of δ. Specifically, we first examine an integer $k \in [\lceil \frac{m}{n-1} \rceil, h^*]$, and

test whether G contains a k-core. If there exists a k-core in G, then we have $\delta \geq k$, based on which we proceed to search in $[k+1, h^*]$; otherwise, we redirect our search to $[\lceil \frac{m}{n-1} \rceil, k-1]$. To determine whether a k-core exists in G, we iteratively remove the nodes in G with degrees smaller than k, until all remaining nodes have degree at least k in the subgraph that they induce. If all nodes in G are removed by this procedure, then G must not contain a k-core; otherwise, the remaining nodes should form a k-core.

Algorithm 1 shows the pseudo-code of SemiDeg. It first computes the h-index h_u for each node u using the Hindex procedure (Lines 2-4). Then, it derives the h-index of $\mathbf{h}$ and uses it as an upper bound of the degeneracy δ (Line 6). Subsequently, it applies the binary search procedure mentioned previously (Lines 7-19). Finally, it returns the degeneracy value δ (Line 20). We illustrate SemiDeg using an example.

Example 2. Consider the graph G in Fig. 1. We have $\mathbf{h} = \{3,3,3,3,2,1,2,1\}$ for the nodes $\{v_1, \ldots, v_8\}$. The h-index of $\mathbf{h}$ equals 3, i.e., $h^* = 3$. On the other hand, we have $\lceil \frac{m}{n-1} \rceil = 2$. Accordingly, SemiDeg sets $lb = 2$ and $ub = 3$ and then performs the binary search procedure on $[2, 3]$. In its first iteration, SemiDeg attempts to find a 2-core (i.e., $mid = 2$) in G by iteratively deleting the nodes with degrees smaller than 2. As a result, SemiDeg obtains a 2-core $\{v_1, v_2, v_3, v_4, v_5, v_7\}$, and records $\delta = 2$. Subsequently, SemiDeg sets $lb = ub = mid = 3$, and tries to find a 3-core in G. This leads to a 3-core $\{v_1, v_2, v_3, v_4\}$, based on which SemiDeg updates δ by setting it to 3. After this step, SemiDeg terminates, and returns $\delta = 3$. □

Theoretical Analysis. The correctness of SemiDeg is guaranteed by Lemma 1. In the following, we analyze the memory overhead and I/O complexity of SemiDeg. Let τ be the maximum number of iterations that SemiDeg requires, for any k, to decide whether a k-core exists in G (see Lines 11-17 in Algorithm 1). We have the following result.

Theorem 1. *The memory, I/O costs, and CPU time complexity of* SemiDeg *are* $O(n)$, $O(\log_2 h^* \times \tau(m+n)/B)$, *and* $O(\log_2 h^* \times \tau(m+n))$ *respectively.*

Proof. SemiDeg only needs to store a constant number of $O(n)$-size arrays in the main memory, and hence, its memory overhead is $O(n)$. For any k, SemiDeg requires $O(\tau(m+n)/B)$ I/Os to determine whether a k-core exists in G. This is because, in each iteration (lines 13-17), the algorithm sequentially scans the *edge file* at most once which takes $O((m+n)/B)$ I/Os in the worst case. Since SemiDeg only examines $O(\log_2 h^*)$ values of k, the total I/O complexity of SemiDeg is $O(\tau(m+n)\log_2 h^*/B)$. Clearly, in each iteration, the algorithm takes $O(d_u(G))$ CPU time to load the nodes $N_u(G)$ from the disk. Therefore, the total CPU time complexity of the algorithm is $O(\log_2 h^* \times \tau(m+n))$. □

Note that both τ and $\log_2 h^*$ are often a small number ($\tau = O(\log n)$ as indicated in [45]). In that case, the I/O complexity of SemiDeg is almost linear to $(m+n)/B$.

Remark. Note that both $h(\mathbf{d})$ (the h-index of the degree vector $\mathbf{d}$) and $\sqrt{2m}$ are well-known upper bounds for the degeneracy. However, both of them are looser than h^* (the upper bound used in Algorithm 1), which leads to more iterations in the binary-search procedure of Algorithm 1, and thus incurs higher I/O costs. We have empirically verified that the performance of the algorithm using $h(\mathbf{d})$ or $\sqrt{2m}$ as upper bounds will be inferior to that of Algorithm 1, which uses h^* as an upper bound.

4.2 The SemiDeg+ Algorithm

Although SemiDeg can compute the degeneracy of G in an I/O-efficient manner, it still suffers from two limitations. First, it requires scanning G once to compute the h-index for each node. When the graph is very large, such a graph scanning procedure can be costly. Second, when deciding whether a k-core exists in G, it requires scanning all nodes with degrees smaller than k as well as the edges associated with those nodes. This procedure may also incur considerable overheads in practice.

To overcome the limitation of SemiDeg, we propose an enhanced algorithm dubbed SemiDeg+. To avoid computing the h-index for every node in G, SemiDeg+ utilizes the h-index of $\mathbf{d}$ (i.e., the set of node degrees in G) as a "cheap" upper bound of the degeneracy δ. More importantly, when testing whether a k-core exists in G, SemiDeg+ applies a novel algorithm (referred to as PCore) that avoids accessing nodes and edges as much as possible. In what follows, we elaborate the PCore algorithm, and then present the details of SemiDeg+.

Algorithm 2. PCore (G, R, $\hat{\mathbf{c}}$, ρ)

Input: $G = (V, E)$ in the disk, the working node set R, upper bounds set $\hat{\mathbf{c}}$, and an integer ρ
Output: The ρ-core R and the updated $\hat{\mathbf{c}}$

```
1: r̂_u ← 0 for all u ∈ V;   /* r̂ is the counting set */
2: update ← 1;
3: while update = 1 do
4:   update ← 0;
5:   for u ∈ R s.t. r̂_u < ρ do
6:     Load N_u(G) from disk;
7:     if ĉ_u = d_u then
8:       ĉ_u ← Hindex (u, ĉ, N_u(G));   /* h-index upper bound */
9:     r̂_u ← |N_u(G) ∩ R|;
10:    if r̂_u < ρ then
11:      R ← R \ {u};   update ← 1;
12:      ĉ_u ← min{ĉ_u, ρ};   /* update the upper bound */
13:      for v ∈ N_u(G) ∩ R do
14:        r̂_v ← r̂_v − 1;
15: return(R, ĉ);
```

The PCore *algorithm.* PCore is based on the following observation.

Observation 1. *If G contains a k-core, the k-core must be in the subgraph induced by the nodes set $R = \{u \mid u \in V, \hat{c}_u \geq k\}$, where $\hat{c}_u$ is a core number upper bound of u.*

Proof. The nodes that are not in R cannot be contained in the k-core, because their core number upper bounds are smaller than k. □

Based on Observation 1, if we are to determine whether G contains a k-core, we only need to consider the subgraph induced by R, denoted as $G(R)$. For convenience, we refer

to R as the *working node set*. The basic idea of PCore is to maintain, for each node $u \in R$, the degree of u in the induced subgraph $G(R)$, and then iteratively deletes the nodes whose degrees in $G(R)$ are smaller than k. Algorithm 2 shows the pseudo-code of PCore.

PCore takes as input G, a positive integer ρ, a set $\hat{\mathbf{c}}$ of core number upper bounds, and a set R of nodes whose core number upper bounds are at least ρ. It returns updated versions of R and $\hat{\mathbf{c}}$, such that (i) $R = \emptyset$ if G does not contain a ρ-core, (ii) otherwise, R is a ρ-core of G. Specifically, PCore uses a set $\hat{\mathbf{r}}$ to maintain the degrees of the nodes in R. Initially, $\hat{r}_u = 0$ for any $u \in R$ (Line 1). Then, for each node $u \in R$ with $\hat{r}_u < \rho$, PCore iteratively loads u's neighbors from the disk (Lines 5-14). If $\hat{c}_u$ equals its original degree in G, PCore updates $\hat{c}_u$ by setting it to the h-index of u w.r.t. the core number upper bounds of u's neighbors, i.e., $h_u(\hat{\mathbf{c}}, N_u(G))$ (Lines 7-8). After that, PCore updates $\hat{r}_u$ to the number of neighbors of u in the working node set R (Line 9). If $\hat{r}_u < \rho$, then u cannot be contained in the ρ-core; in that case, PCore removes u from R (Line 11), and also updates $\hat{c}_u$ to ρ (Line 12), since the core number of u must be smaller than ρ. Subsequently, for each neighbor v of u in the working node set R, PCore updates $\hat{r}_v$ (Lines 13-14). Finally, PCore returns R and $\hat{\mathbf{c}}$ (Line 15). The following example illustrates how PCore works.

Example 3. Consider the graph in Fig. 1. Suppose $R = \{v_2, v_3, v_4, v_5\}$, $\hat{\mathbf{c}} = \{3, 4, 5, 4, 4, 1, 2, 1\}$ for the nodes $\{v_1, \ldots, v_8\}$, and $\rho = 4$. First, PCore loads v_2's neighbors from the disk, and computes the h-index of v_2, which is equal to 3 (lines 7-8 in Algorithm 2). Then, PCore updates $\hat{r}_{v_2}$ by 3, as v_2 has three neighbors in R. Since $\hat{r}_{v_2} < \rho = 4$, PCore deletes v_2 from R. Second, PCore loads v_3's neighbors from the disk, and updates $\hat{c}_{v_3}$ by $h_{v_3}(\hat{\mathbf{c}}, N_{v_3}(G))$ which equals 3. Then, PCore updates $\hat{r}_{v_3}$ by 2, as v_3 has two neighbors in R ($R = \{v_3, v_4, v_5\}$). PCore also removes v_3 from R, because $\hat{r}_{v_3} < \rho$. Similarly, we can easily derive that PCore also deletes v_4 and v_5, and updates $\hat{c}_{v_4}$ and $\hat{c}_{v_5}$ by 3 and 2 respectively. □

The following theorem shows the correctness of PCore.

Theorem 2. *If G contains a ρ-core, then* PCore *returns the ρ-core and a correct upper bound set $\hat{\mathbf{c}}$.*

Proof. Let R and R^* be the input and output working node set of PCore, respectively. First, we show that if $R^* \neq \emptyset$, then R^* is the ρ-core in G. This is because when PCore terminates, $\hat{r}_u \geq \rho$ for each $u \in R^*$. Thus, the nodes in R^* satisfy the degree constraint of the ρ-core. To show that R^* is the maximal subset satisfying such a degree constraint, we assume to the contrary that there is a superset $\tilde{R}$ of R^* that also satisfies the degree constraint of the ρ-core. Since R contains the ρ-core, we have $\tilde{R} \subseteq R$. As a consequence, there is a node $u \in \tilde{R}$ and $u \notin R^*$ that is deleted by PCore. In that case, we have $\hat{r}_u < \rho$, which contradicts to the assumption that $\tilde{R}$ satisfies the degree constraint.

Second, $h(\hat{\mathbf{c}}, N_u(G))$ is a valid upper bound of c_u. If a node u is removed PCore, we have $c_u < \rho$. Thus, the upper bound updating strategies of PCore (Lines 8 and 12 in Algorithm 2) is correct. As a result, PCore correctly outputs a refined upper bounds set. □

Details of SemiDeg+. We present the details of SemiDeg+ in Algorithm 3. The algorithm first computes $\lceil \frac{m}{n-1} \rceil$ and $h(\mathbf{d})$ as the initial lower and upper bounds of δ, respectively (Lines 1-3). After that, it performs an iteratively-halving procedure to tighten lower and upper bounds of δ, and to obtain a 2-approximation of δ (Lines 4-10). In each iteration of the procedure, the algorithm considers a working node set $R = \{u \mid u \in V, \hat{c}_u \geq ub\}$ (Line 5), and invokes PCore determine whether a ub-core exists. After the iterative procedure terminates, the algorithm performs a binary search over the interval $[lb, ub]$ to compute the exact value of δ, using PCore in each iteration (Lines 11-16). We illustrate the algorithm using the an example.

Example 4. Consider the graph in Fig. 1. First, we have $\hat{\mathbf{c}} = \{3, 4, 5, 4, 4, 1, 2, 1\}$ for the nodes $\{v_1, \ldots, v_8\}$. Clearly, we have $lb = \lceil \frac{m}{n-1} \rceil = 2$ and $h(\mathbf{d}) = 4$. In the iteratively-halving procedure (Lines 4-10), SemiDeg+ first invokes PCore with a working node set $R = \{v_2, v_3, v_4, v_5\}$ and upper bounds set $\hat{\mathbf{c}}$ to identify whether a 4-core exists. As shown in the Example 3, PCore would return $R = \emptyset$ and $\hat{\mathbf{c}} = \{3, 3, 3, 3, 2, 1, 2, 1\}$. Then, SemiDeg+ halves the upper bound to $h(\mathbf{d})/2 = 2$, and invokes PCore with inputs $R = \{v_1, v_2, v_3, v_4, v_5, v_7\}$, $\hat{\mathbf{c}} = \{3, 3, 3, 3, 2, 1, 2, 1\}$, and $\rho = 2$. It can be verified that PCore returns $R = \{v_1, v_2, v_3, v_4, v_5, v_7\}$ and $\hat{\mathbf{c}} = \{3, 3, 3, 3, 2, 1, 2, 1\}$. Since there is a 2-core, SemiDeg+ terminates the iteratively-halving procedure.

After that, SemiDeg+ performs a binary search over the interval $[2, 4]$. First, we have $mid = 3$, and thus, SemiDeg+ invokes PCore with inputs $R = \{v_2, v_3, v_4, v_5\}$, $\hat{\mathbf{c}} = \{3, 3, 3, 3, 2, 1, 2, 1\}$, and $mid = \rho = 3$. Accordingly, PCore returns $R = \{v_2, v_3, v_4, v_5\}$ as a 3-core and keeps $\hat{\mathbf{c}}$ unchanged. Then, SemiDeg+ records $\delta = 3$ (Line 14), and updates $lb = 4$. Subsequently, SemiDeg+ invokes PCore with inputs $R = \emptyset$, $\hat{\mathbf{c}} = \{3, 3, 3, 3, 2, 1, 2, 1\}$, and $mid = \rho = 4$. Since $R = \emptyset$, PCore immediately terminates without incurring any I/O cost. Then, SemiDeg+ updates $ub = mid - 1 = 3$. Since $ub < lb$, SemiDeg+ terminates and returns $\delta = 3$ as the final result. □

Analysis of SemiDeg+. The correctness of SemiDeg+ directly follows the correctness of PCore, which is shown in Theorem 2. In the following, we analyze the memory and I/O overheads of SemiDeg+. Let τ be the maximum number of iterations needed in PCore to compute whether the working node set R contains a k-core, $\tilde{n}$ be the maximum number of nodes in R, and $\tilde{m}$ be the total number of incident edges of the nodes in R. We have the following result.

Theorem 3. *The memory, I/O costs, and CPU time complexity of* SemiDeg+ *are $O(n)$, $O(\log_2 h(\mathbf{d}) \times \tau(\tilde{n} + \tilde{m})/B)$, and $O(\log_2 h(\mathbf{d}) \times \tau(\tilde{n} + \tilde{m}))$ respectively.*

Proof. SemiDeg+ only needs to maintain two $O(n)$ size arrays, i.e., $\hat{\mathbf{r}}$ and $\hat{\mathbf{c}}$, as well as the working node set R. Therefore, the memory cost of SemiDeg+ is $O(n)$. As for the I/O cost of SemiDeg+, we observe that in Lines 1-10 in Algorithm 3, SemiDeg+ has at most $O(\log_2 h(\mathbf{d}))$ iterations, which incurs at most $O(\log_2 h(\mathbf{d}) \times \tau(\tilde{n} + \tilde{m})/B)$ I/Os. Let $[lb, ub]$ be the binary-search interval in Lines 11-15. The number of iterations required for a binary search on

$[lb, ub]$ is $O(\log_2(ub - lb)) = O(\log_2 \delta) \leq O(\log_2 h(\mathbf{d}))$, since $lb \leq \delta \leq ub \leq 2 \times lb$. As a result, the total number of I/Os of SemiDeg+ is $O(\log_2 h(\mathbf{d}) \times \tau(\tilde{n} + \tilde{m})/B)$. Similarly, we can easily derive that the CPU time complexity of SemiDeg+ is $O(\log_2 h(\mathbf{d}) \times \tau(\tilde{n} + \tilde{m}))$, because the load-neighobrhood operator takes $O(d_u(G)$ time for each u. □

Comparison With Other Algorithms. Compared with SemiDeg, SemiDeg+ has the following advantages. First, SemiDeg+ only works on a small working node set R, which leads to much higher efficiency. Second, SemiDeg+ does not compute the h-index for every node, but only derive the h-index for a node on-demand, which significantly reduces the number of I/Os. The reason is that in an iteration, computing the h-index for all nodes takes $O((m+n)/B)$ I/Os, while SemiDeg+ only calculates the h-index for the nodes that are contained in R and also meet the constraint $\hat{r}_u < \rho$ (see lines 5-8 in Algorithm 2), thus the I/O costs can be much lower than $O((m+n)/B)$.

Algorithm 3. SemiDeg+ (G)

Input: $G = (V, E)$ in the disk
Output: The degeneracy δ of G
1: Let d_u be the degree of $u \in V$; $\hat{c}_u \leftarrow d_u$ for each $u \in V$;
2: Let $u_{\max}$ be the node that has the largest degree in G;
3: $lb \leftarrow \sum_{u \in V} d_u/2(n-1)$; $ub \leftarrow$ Hindex ($u_{\max}$, $\mathbf{d}$, V);
4: **while** $h_d \geq lb$ **do**
5: $R \leftarrow \{u \,|\, u \in V, \hat{c}_u \geq ub\}$;
6: $(C, \hat{\mathbf{c}}) \leftarrow$ PCore (G, R, $\hat{\mathbf{c}}$, ub);
7: **if** $C \neq \emptyset$ **then break**;
8: **else** $ub \leftarrow ub/2$; ;/* halve the upper bound */
9: **if** $ub \geq lb$ **then** $lb \leftarrow ub$;
10: $ub \leftarrow 2 \cdot ub$; /* ub is a 2-approximation of degeneracy */
11: **while** $lb \leq ub$ **do**
12: $mid \leftarrow \lfloor (lb + ub)/2 \rfloor$; $R \leftarrow \{u \,|\, u \in V, \hat{c}_u \geq mid\}$;
13: $(C, \hat{\mathbf{c}}) \leftarrow$ PCore (G, R, $\hat{\mathbf{c}}$, mid);
14: **if** $C \neq \emptyset$ **then** $lb \leftarrow mid + 1$; $\delta \leftarrow mid$;
15: **else** $ub \leftarrow mid - 1$;
16: **return** δ;

Compared with SemiCore [33], SemiDeg+ excels in efficiency because (i) SemiCore needs to compute all k-cores before obtaining the degeneracy δ, which incurs considerable I/O costs, and (ii) SemiDeg+ only derives a small number of k-cores in its derviation of δ, which is much more efficient.

Note that SemiDeg+ can also return a 2-approximate degeneracy value when the iteratively-halving procedure terminates (Lines 4-10). This approximate version of SemiDeg+ (i.e., the iteratively-halving procedure) is not only much more efficient than SemiStream [12], but it also achieves better approximate ratio than SemiStream, as demonstrated in Section 6.

Discussions. Any ordering of nodes in an undirected graph $G = (V, E)$ can generate a directed graph G' with the same nodes, in which each edge is oriented from the high-order node in the low-order one. The degeneracy ordering is an ordering such that the maximum out-degree of the node in the yielded directed graph G' is no larger than δ [12], [36]. Note that after obtaining the degeneracy δ, it is straightforward to compute the degeneracy ordering by iteratively removing the nodes with degrees smaller than δ.

5 Degeneracy Maintenance

In this section, we show how to incrementally maintain the degeneracy under the semi-external setting, given that the graph is updated by an edge insertion or deletion. Obviously, we can apply the SemiCore algorithm to maintain the degeneracy. SemiCore, however, is inefficient for degeneracy maintenance, because it has to maintain all the core numbers of nodes when an edge is updated. Intuitively, an efficient degeneracy maintenance algorithm should only maintain the $c_{\max}$-core, as the degeneracy has nothing to do with other k-cores for $k < c_{\max}$. The key issue is how can we efficiently maintain the $c_{\max}$-core without maintaining the other k-cores.

Note that in our problem, the challenges that we face are fundamentally different from the traditional k-core maintenance problem. This is because in our problem, we only have the core numbers of the nodes in the $c_{\max}$-core, and no core number is provided for the other nodes. Therefore, the traditional core maintenance techniques [33], [46], [47], which need to know all core numbers, cannot be used for our problem. Below, we develop a novel $c_{\max}$-core maintenance approach based on the PCore algorithm to tackle this challenge.

5.1 Handling Edge Deletion

We first consider the edge deletion case. Let (u, v) be an edge to be deleted. Recall that by Algorithm 3, we can obtain the degeneracy $c_{\max}$, the $c_{\max}$-core denoted by $C_{\max}$, as well as the degree set of nodes, i.e., $\mathbf{d} = \{d_{v_1}, \ldots, d_{v_n}\}$. Clearly, to maintain the degeneracy, it is sufficient to maintain the $c_{\max}$-core $C_{\max}$. By the result shown in [46], $C_{\max}$ may be updated only if both u and v are contained in $C_{\max}$. Thus, in the following, we only consider the case when both $u \in C_{\max}$ and $v \in C_{\max}$.

Let $\hat{r}_u$ be the number of neighbors of u in $C_{\max}$, i.e., $\hat{r}_u = |N_u(G) \cap C_{\max}|$. We have the following result.

Lemma 2. *After deleting (u, v), $C_{\max}$ will be updated only if $\hat{r}_u < c_{\max}$ or $\hat{r}_v < c_{\max}$.*

Proof. If $\hat{r}_u < c_{\max}$ ($\hat{r}_v < c_{\max}$), we know that u (v) has less than $c_{\max}$ neighbors in $C_{\max}$, thus its core number must decrease by 1. Therefore, we must delete the node u (v) from $C_{\max}$. On the other hand, if both $\hat{r}_u \geq c_{\max}$ and $\hat{r}_v \geq c_{\max}$, u and v are still contained in $C_{\max}$ by the definition of the $c_{\max}$-core. □

By Lemma 2, if $\hat{r}_u < c_{\max}$ or $\hat{r}_v < c_{\max}$, we can invoke PCore to maintain the $c_{\max}$-core. Recall that PCore admits three input parameters: the working node set, the upper bounds set, and the parameter ρ. We can use $C_{\max}$ as the working node set, since it must contain the updated $c_{\max}$-core. We update the degrees d_u and d_v after removing (u, v), and make use of the updated degree sets $\mathbf{d}$ as the upper bounds set. For the parameter ρ, we set it to $c_{\max}$. Clearly, we can obtain a $c_{\max}$-core, if it exists, by invoking PCore with these parameters. Note that PCore may return an empty set if the $c_{\max}$-core does not exist. In this case, the entire $c_{\max}$-core is vanished after deleting (u, v). Thus, we has to compute the $(c_{\max} - 1)$-core, as an edge deletion can only decrease the maximum core number (degeneracy) by 1

based on the result shown in [46]. Again, we are able to apply the PCore algorithm to compute $(c_{\max}-1)$-core. It is important to note that the updated $(c_{\max}-1)$-core may contains the original $c_{\max}$-core. Therefore, we cannot use $C_{\max}$ as the working node set. Instead, we set $R=\{u|u\in V, d_u \geq c_{\max}-1\}$, because R obviously contains the $(c_{\max}-1)$-core. Also, we set the updated degree set as the upper bounds set, and $\rho = c_{\max}-1$. The detailed implementation of our algorithm is depicted in Algorithm 4.

Example 5. Consider the graph in Fig. 1. Suppose that we delete an edge (v_1, v_2). Clearly, before deleting (v_1, v_2), we have $\mathbf{d}=\{3,4,5,4,4,1,2,1\}$, $c_{\max}=3$, and $C_{\max}=\{v_1, v_2, v_3, v_4\}$. First, the algorithm updates $d_{v_1}=2$ and $d_{v_2}=3$. Then, the algorithm calculates $\hat{r}_{v_1}=2$ and $\hat{r}_{v_2}=2$, because both $v_1\in C_{\max}$ and $v_2\in C_{\max}$ (Lines 2-3). Since $\hat{r}_{v_1}<c_{\max}$, the algorithm invokes PCore to compute the $c_{\max}$-core (Lines 4-5). We can easily derive that PCore returns $\emptyset$, as there is no 3-core after deleting (v_1, v_2). Thus, the algorithm computes the $(c_{\max}-1)$-core by using the working node set $R=\{u|u\in V, d_u\geq 2\}=\{v_1,\ldots,v_5,v_7\}$ (Lines 6-8). PCore will return R as the $(c_{\max}-1)$-core, and the Deletion algorithm updates $c_{\max}$ by $C_{\max}$ accordingly (Lines 6-8). □

Analysis of Deletion. The correctness of Algorithm 4 can be guaranteed by Lemma 2 and Theorem 2. Clearly, the memory overhead of Algorithm 4 is $O(n)$. Below, we mainly analyze the I/O complexity and the CPU time complexity of Algorithm 4. Let τ be the number of iterations taken by PCore, $\tilde{n}$ be the number of nodes in the working node set R, and $\tilde{m}$ be the total number of incident edges of the nodes in R.

Algorithm 4. Deletion (G, (u,v), $\mathbf{d}$, $c_{\max}$, $C_{\max}$)

Input: Graph G, edge (u,v), degree set $\mathbf{d}$, $c_{\max}$, and the $c_{\max}$-core $C_{\max}$
Output: The updated degeneracy $c_{\max}$, $c_{\max}$-core $C_{\max}$, and $\mathbf{d}$
1: Update d_u and d_v after removing edge (u,v);
2: **if** $u\in C_{\max}$ and $v\in C_{\max}$ **then**
3: $\hat{r}_u \leftarrow |N_u(G)\cap C_{\max}|$; ; $\hat{r}_v \leftarrow |N_v(G)\cap C_{\max}|$;
4: **if** $\hat{r}_u < c_{\max}$ or $\hat{r}_v < c_{\max}$ **then**
5: $(C_{\max}, \hat{\mathbf{c}}) \leftarrow$ PCore (G, $C_{\max}$, $\mathbf{d}$, $c_{\max}$);
6: **if** $C_{\max}=\emptyset$ **then**
7: $c_{\max} \leftarrow c_{\max}-1$;
8: $(C_{\max}, \hat{\mathbf{c}}) \leftarrow$ PCore (G, $\{u|u\in V, d_u\geq c_{\max}\}$, $\mathbf{d}$, $c_{\max}$);
9: **return** ($\mathbf{d}$, $c_{\max}$, $C_{\max}$);

Theorem 4. *To handle an edge (u,v), the I/O and CPU time complexity of Algorithm 4 is $O(\tau(\tilde{m}+\tilde{n})/B)$ and $O(\tau(\tilde{m}+\tilde{n}))$ respectively, if the $c_{\max}$-core is updated. Otherwise, the I/O and CPU time complexity is $O((d_u+d_v)/B)$ and $O((d_u+d_v))$ respectively.*

Proof. Clearly, if the $c_{\max}$-core is not updated, Algorithm 4 only needs to update d_u and d_v, as well as compute $\hat{r}_u$ and $\hat{r}_v$, which can be done by loading the neighbors of u and v from the disk once. Thus, in this case, the I/O and CPU time complexity are $O((d_u+d_v)/B)$ and $O(d_u+d_v)$ respectively. If the $c_{\max}$-core is updated, Algorithm 4 has to invoke PCore to maintain the $c_{\max}$-core, thus its I/O and CPU time complexity are the same as those of PCore, which are $O(\tau(\tilde{m}+\tilde{n})/B)$ and $O(\tau(\tilde{m}+\tilde{n}))$ respectively. □

In the experiments, we show that our algorithm is very efficient in practice, because the $c_{\max}$-core is updated infrequently even when the graph is frequently updated. On the other hand, the number of iterations taken by PCore to compute the $c_{\max}$-core can be bounded by $O(\log n)$ in random graphs, as indicated in [45]. Thus, even if the $c_{\max}$-core is updated, the I/O complexity of our algorithm is expected to be bounded by $O(\log n \times (\tilde{m}+\tilde{n})/B)$.

5.2 Handling Edge Insertion

Here we discuss the edge insertion case. Let (u,v) be an edge to be inserted. The algorithm first updates the degrees d_u and d_v after adding (u,v). Then, it is easy to show that $C_{\max}$ may be updated only if both $d_u\geq c_{\max}$ and $d_v\geq c_{\max}$. To further improve the efficiency, we can compute the h-index of u (v), denoted by h_u (h_v), based on the updated degrees. Based on the h-index, we can derive the following result.

Lemma 3. *After inserting (u,v), $C_{\max}$ cannot be update if $h_u < c_{\max}$ or $h_v < c_{\max}$.*

Proof. Suppose, without loss of generality, that $h_u < c_{\max}$. Then, we have $c_u < c_{\max}$, as h_u is an upper bound of c_u. Clearly, u does affect $C_{\max}$, and the number of neighbors of v in $C_{\max}$ also keeps unchanged. As a result, no node's core number will be updated in this case. □

By Lemma 3, we only need to maintain the $c_{\max}$-core when both $h_u\geq c_{\max}$ and $h_v\geq c_{\max}$. Below, we assume that $h_u\geq c_{\max}$ and $h_v\geq c_{\max}$, and consider two cases. First, if both $u\in C_{\max}$ and $v\in C_{\max}$, the $c_{\max}$-core may contain a $(c_{\max}+1)$-core after adding (u,v). Thus, we invoke PCore with working node set $R=C_{\max}$, upper bounds set $\mathbf{d}$, and $\rho=c_{\max}+1$ to compute the $(c_{\max}+1)$-core. If such a $(c_{\max}+1)$-core exists, we update $C_{\max}$ by the $(c_{\max}+1)$-core, and increase $c_{\max}$ by 1. Otherwise, we keep both $c_{\max}$ and $C_{\max}$ unchanged, because both u and v are already in $C_{\max}$ and thereby the insertion of (u,v) does not affect $C_{\max}$. Second, if there exist at least one node of u and v that are not in $C_{\max}$, we invoke PCore with parameters $R=\{u|u\in V, d_u\geq c_{\max}\}$, d, and $\rho=c_{\max}$ to compute the $c_{\max}$-core. This is because under this case, the $c_{\max}$-core may be expanded after inserting an edge (u,v), and therefore we need to invoke PCore to recompute the $c_{\max}$-core. Moreover, in this case, the $c_{\max}$-core does not contain a $(c_{\max}+1)$-core. The detailed implementation of our algorithm is given in Algorithm 5.

Example 6. Consider the graph in Fig. 1. Suppose that we have already deleted the edge (v_1, v_2), and we aim to maintain the degeneracy after adding back (v_1, v_2). Clearly, by Example 5, we have $\mathbf{d}=\{2,3,5,4,4,1,2,1\}$, $c_{\max}=2$, and $C_{\max}=\{v_1,\ldots,v_5,v_7\}$ for the graph in Fig. 1 after deleting (v_1, v_2). When inserting back (v_1, v_2), the algorithm first updates $d_{v_1}=3$ and $d_{v_2}=4$ (Line 1 in Algorithm 5). Since both $d_{v_1}\geq c_{\max}$ and $d_{v_2}\geq c_{\max}$, the algorithm computes $h_{v_1}=3$ and $h_{v_2}=3$ (Lines 2-3). Then, since (i) $h_{v_1}\geq c_{\max}$ and $h_{v_2}\geq c_{\max}$, and (ii) both $v_1\in C_{\max}$ and $v_2\in C_{\max}$, the algorithm invokes PCore with parameters $R=C_{\max}$, $\mathbf{d}$, and $\rho=3$ to compute the 3-core (Lines 4-6). Clearly, the algorithm is able to obtain a 3-core $\{v_1,\ldots,v_4\}$. Thus, the algorithm updates $C_{\max}$ by this 3-core, and sets $c_{\max}=3$ (Lines 7-8). □

Analysis of Insertion. The correctness of Algorithm 5 can be guaranteed by Lemma 3 and Theorem 2. Similar to Algorithm 4, the memory overhead of Algorithm 5 is $O(n)$. The I/O and CPU time complexity of Algorithm 4 are $O(\tau(\tilde{m}+\tilde{n})/B)$ and $O(\tau(\tilde{m}+\tilde{n}))$ respectively, if both $h_u \geq c_{\max}$ and $h_v \geq c_{\max}$ after inserting (u,v). Otherwise, the I/O and CPU time complexity are $O((d_u+d_v)/B)$ and $O(d_u+d_v)$ respectively. Since $C_{\max}$ is infrequently update even when the graph is rapidly changed, Algorithm 5 is very efficient in practice, as confirmed in our experiments.

Algorithm 5. Insertion (G, (u,v), $\mathbf{d}$, $c_{\max}$, $C_{\max}$)

Input: Graph G, edge (u,v), degree set $\mathbf{d}$, $c_{\max}$, and the $c_{\max}$-core $C_{\max}$
Output: The updated degeneracy $c_{\max}$, $c_{\max}$-core $C_{\max}$, and $\mathbf{d}$

```
1: Update d_u and d_v after inserting edge (u, v);
2: if d_u ≥ c_max and d_v ≥ c_max then
3:   h_u ← Hindex(u, d, N_u(G)); h_v ← Hindex(v, d, N_v(G));
4:   if h_u ≥ c_max and h_v ≥ c_max then
5:     if u ∈ C_max and v ∈ C_max then
6:       (C, ĉ) ← PCore (G, C_max, d, c_max + 1);
7:       if C ≠ ∅ then
8:         c_max ← c_max + 1; C_max ← C;
9:     else
10:      (C_max, ĉ) ← PCore (G, {u|u ∈ V, d_u ≥ c_max}, d, c_max);
11: return(d, c_max, C_max);
```

6 Experiments

In this section, we first conduct extensive experiments to evaluate the efficiency of the proposed algorithms. Then, we systematically evaluate the degeneracies of 150 publicly available real-world networks.

6.1 Experimental Setup

We collect 150 various real-world networks from four different sources, including (1) the Koblenz Network Collection (http://konect.uni-koblenz.de/), (2) the Stanford Network Collection (http://snap.stanford.edu/data/), (3) the Web Graph Collection (http://webgraph.di.unimi.it/), and the ASU Network Collection (http://socialcomputing.asu.edu/pages/datasets). The detailed statistics of these networks are shown in Table 2. Note that the original GSH dataset released at http://webgraph.di.unimi.it/is very large which takes near 1TB after decompressing. Due to the hardware limit, our GSH dataset in Table 2 is a subgraph generated by randomly sampling edges from the original GSH graph.

We implement six various algorithms: SemiStream, Sampling, SemiCore, SemiDeg, SemiDeg+, and SemiDegAppr. SemiStream is the semi-streaming approximate algorithm proposed in [12] (similar idea was also proposed in [13] and [34]). For SemiStream, we set the parameter $\alpha = 4$ to achieve good I/O performance. Sampling is a one-pass streaming algorithm [14] which is based on a carefully-designed sampling technique. Recall that Sampling uses $O(\frac{n(\log_2 n)}{\epsilon^2})$ space. In our experiments, we set $\epsilon = 2$ to ensure that the algorithm uses around $O(n)$ memory (e.g., if $\epsilon = 2$ and $\log_2 n = 32$, then $\frac{n(\log_2 n)}{\epsilon^2} = 5n/4$). SemiCore denotes the state-of-the-art semi-external core decomposition algorithm [33]. SemiDeg and SemiDeg+ denote Algorithm 1 and Algorithm 3 respectively. SemiDegAppr is essentially the iteratively-halving procedure in Algorithm 3 which can generate a 2-approximate solution of the degeneracy.

Experimental Settings. All algorithms are implemented in C++, using gcc compiler with no compilation flag. All experiments are conducted on a PC with a 2.4GHz Xeon CPU, DDR4 2400 MHZ memory (16GB), and 7200 RPM SATA III 1 TB disk with 600MB/s data transfer rate, running Red Hat Linux 6.4. We conduct each experiment independently on this PC, and thus any two experiments do not compete for resources. For all experiments, the time cost of each algorithm is measured by the amount of wall-clock time elapsed during the algorithms' execution. For each input graph G, we organize G in the disk using the graph storage method described in Section 2. In addition, each node's adjacency list is sorted by the nodes' IDs using a standard external-memory sorting algorithm. For all our algorithms, we only store the node information (e.g., the core number upper bounds $\mathbf{c}$) in the main memory. For the memory costs, we record the maximum amount of memory used by each algorithm during the algorithms' execution. Recall that when the algorithm visits the neighborhood of a node, it needs to load the adjacency list of that node from the disk, thus incurring I/O costs. We make use of the standard method as used in [33], [38] to record the number of I/Os for various algorithms.

6.2 Performance Studies

We evaluate the performance of different I/O-efficient algorithms for degeneracy measurement and maintenance using two sets of networks: 1) five medium-sized graphs which are ctPaTe, LiveJour, Hollywood, Orkut, and Arabic; and 2) five massive graphs, including IT, Twitter, SK, UK, and GSH. ctPaTe is a citation network, and Hollywood is a co-actor network. LiveJour, Orkut, and Twitter are social networks. Arabic, IT, SK, UK, and GSH are web graphs. The detailed statistics of these networks are shown in Table 2 (in bold font).

Results for Degeneracy Computation. Fig. 2 reports the running time, I/O cost, and memory overhead of various algorithms for degeneracy computation. As shown in Figs. 2a and 2b, Sampling is the fastest algorithm on most datasets, followed by SemiDegAppr, SemiDeg+, SemiStream, SemiCore, and SemiDeg. We can also observe that Sampling is slightly faster than SemiDegAppr, but it cannot work on the largest GSH dataset due to its high memory usage. Generally, SemiDegAppr is nearly 2 times faster than SemiDeg+, and SemiDeg+ is around one order of magnitude faster than SemiCore. We can also observe that SemiDeg+ is significantly faster than SemiStream in massive graphs. For example, on the largest network GSH, SemiDegAppr takes 385 seconds, SemiDeg+ consumes 633 seconds, SemiStream takes 2,808 seconds, and SemiCore uses 7,218 seconds to compute the degeneracy. It is worth mentioning that SemiDeg is not very efficient, since it needs to scan a large portion of the graph in each iteration. Thus, for massive graphs, we do not show the results of SemiDeg. Figs. 2c 2d show the running time of different algorithms plus the preprocessing time (i.e., the sorting cost). On the medium-sized graphs, the results are consistent with the results shown in Figs. 2a and 2b. On the massive graphs, however, most algorithms except SemiCore achieve similar performance, because the preprocessing time often dominates the time for computing the degeneracy.

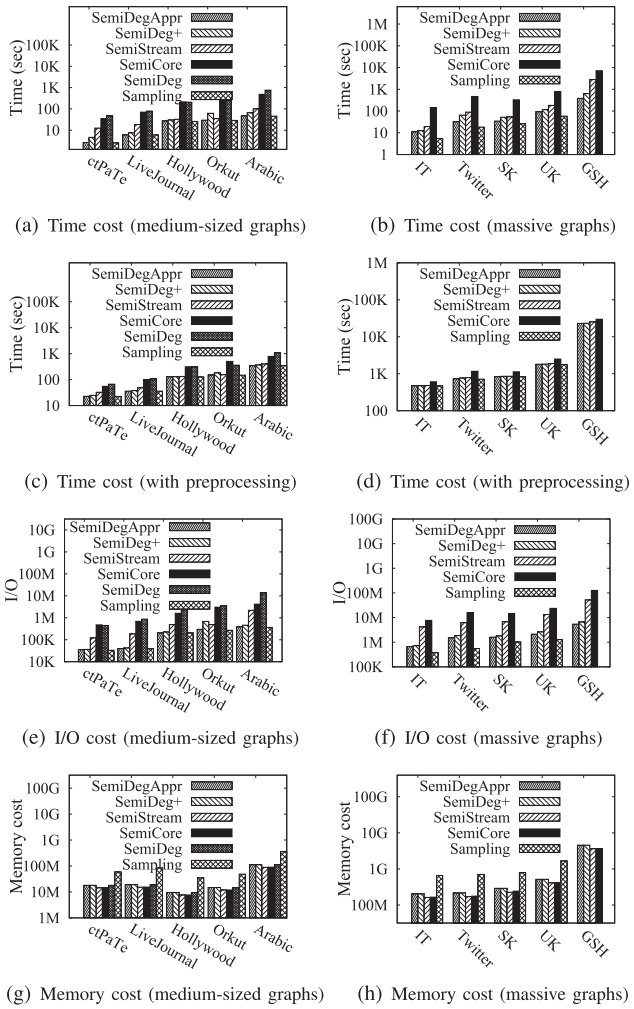
TABLE 2
Networks Statistics and the Degeneracy Results (1K=1,000, 1M=1,000,000, and 1G=1,000,000,000)

Networks	Name	$\|V\|$	$\|E\|$	δ	Name	$\|V\|$	$\|E\|$	δ	Name	$\|V\|$	$\|E\|$	δ
Citation networks	ctDBLP	12.6K	99.5K	12	ctCora	23K	183K	13	ctHeTH	28K	706K	37
	ctHePH	35K	843K	30	ctSeer	384.4K	3.5M	15	ctPaTe	**3.8M**	**33M**	**64**
Collaboration networks	caGrQc	5.2K	14.5K	43	caHeTH	9.8K	26K	31	caHePH	12K	118.5K	238
	caAsPh	18.8K	198K	56	caCoMa	23K	93.5K	25	caDBLP	933K	13M	118
Infrastructure networks	RoadEU	1.2K	1.4K	2	AirTra	1.2K	2.6K	4	RoadCH	1.5K	1.3K	1
	USAir	1.6K	56.5K	64	OPFlight	2.9K	61K	28	PowGrid	4.9K	6.6K	5
	RoadPA	1.1M	1.5M	3	RoadTX	1.4M	1.9M	3	RoadCA	2M	2.8M	3
Biology networks	Elegans	453	4.6K	10	Stelzl	1.7K	12.4K	7	Protein	1.9K	2.3K	5
	Figeys	2.2K	12.9K	10	Vidal	3.1K	6.7K	6	Reactome	6.3K	147.5K	176
Software networks	JungDep	6.1K	138.7K	65	JDK	6.4K	151K	65	LinuxSC	30.8K	214K	23
Lexical networks	Thesaurus	23.1K	511.8K	34	WordNet	146K	657K	31	YahooAdv	653.3K	2.9M	39
Computer networks	AsRoute	6.5K	14K	12	Oregon	11.5K	65.4K	31	NetTO	34.8K	171.4K	63
	Caida	26.5K	53.4K	22	Gnutella	62.6K	295.8K	6	Skitter	1.7M	11.1M	111
P2P networks	P2PGnu04	10.9K	80K	3	P2PGnu05	8.8K	63.7K	3	P2PGnu06	8.7K	63.1K	3
	P2PGnu08	6.3K	41.6K	3	P2PGnu09	8.1K	52K	3	P2PGnu24	26.5K	130.7K	2
	P2PGnu25	22.7K	109.4K	2	P2PGnu30	36.7K	176.7K	2	P2PGnu31	62.6K	295.8K	2
Communication networks	RoviraU	1.1K	5.4K	11	UCIrvine	1.9K	59.8K	20	DNCEmail	2K	39.3K	17
	DiggCom	30.4K	87.6K	9	Enron	36.7K	183.8K	43	FBWall	47K	877K	16
	Slashdot	51.1K	140.8K	14	LinuxMail	63.4K	1.1M	91	WikiDE	91.3K	2.4M	117
	WikiDU	225.7K	1.55M	98	EmailEU	256K	420K	37	37 WikiRU	457K	2.3M	81
	WikiSP	497.4K	2.7MK	94	WikiGE	519.4K	6.7M	150	WikiPO	541.3K	2.4M	84
	WikiIT	863.8K	3.1M	107	WikiArab	1.1M	1.9M	54	WikiCN	1.2M	2.3M	68
	WikiFR	1.4M	4.64M	120	WikiTalk	2.4M	5M	131	WikiEN	3M	2.5M	210
Online contact networks	EmailCon	2K	136.6K	74	WikiElect	7.1K	103.7K	53	PreGood	10.7K	24.3K	31
	WikiConf	118.1K	2.9M	145	WikiSign	138.6K	740.4K	55	UbuntuCon	159.3K	964.4K	48
Hyperlink networks	BlogElect	1.2K	19K	36	Foldoc	13.3K	125.2K	9	GoogleIn	15.8K	171.2K	102
	WebStanf	282K	2.3M	71	WebND	325.7K	1.5M	155	ReBaidu	415.6K	3.3M	228
	WebBerke	685.2K	7.6M	201	WebGoogle	875.7K	5.1M	44	LKWikiPLD	1M	25M	225
	LKWikiNL	1M	20.1M	166	LKWikiITD	1.2M	34.8M	271	LKWikiPL	1.5M	57.5M	842
	TrecWT	1.6M	8.1M	140	LKWikiPT	1.6M	49M	1043	LKWikiJA	1.6M	71.1M	848
	LKWikiIT	1.9M	91.6M	857	LKHuDong	2M	14.9M	266	LKBaidu	2.1M	17.8M	78
	LKWikiFRD	2.2M	59M	221	ReHuDong	2.4M	18.9M	16	LKWikiRU	2.9M	82M	818
	LKWikiFR	3M	102.4M	1097	LKWikiGE	3.2M	81.6M	829	WebIndo	7.4M	194.1M	6869
	EUHost	11.3M	386.9M	987	LKWikiEN	12.1M	378.1M	1123	LKDBPedia	18.3M	172.2M	149
	Arabic	**22.7M**	**640M**	**3,247**	EUTpd	6.65M	170.1M	9,874	IT	**41.3M**	**1.2G**	**3,224**
	SK	**50.6M**	**1.95G**	**4,510**	UK	**106.3M**	**3.87G**	**10,424**	GSH	**930M**	**13.3G**	**3,954**
Social networks	HamsterF	1.9K	12.5K	20	HamsterA	2.4K	16.6K	24	FbEgo	2.9K	3K	3
	Advogato	6.5K	51.1K	25	WikiVote	7.1K	103.7K	53	Google+	23.6K	39.2K	12
	Brightki	58.2K	214.1K	52	FbSocial	63.7K	817K	52	Epinions	75.9K	508.9K	67
	Slashdot	79.1K	515.4K	54	BlogCal	88.8K	4.2M	221	BlogCall	97.9K	2M	220
	Buzznet	101.2K	4.3M	153	Deliciou	103.1K	1.4M	33	Livemoch	104.1K	2.2M	92
	Foursqua	106.2K	3.5M	63	LastFM	108.5K	5.1M	70	EPTrust	131.8K	841.4K	121
	CatstFri	149.7K	5.4M	419	Douban	154.9K	327.2K	15	Gowalla	196.6K	950.3K	51
	Libimset	221K	17.4M	273	DiggFri	279.6K	1.7M	176	DogstFri	426.8K	8.5M	248
	FamiLink	623.8K	15.7M	1,159	Youtulink	1.14M	4.94M	51	Hyves	1.4M	2.8M	39
	Pokec	1.6M	30.6M	47	FlickrSocI	1.7M	15.6M	568	FlickrSocII	2.3M	33.1M	600
	Flixster	2.5M	7.9M	68	Orkut	**3.1M**	**117.2M**	**253**	YouTube	3.2M	9.4M	51
	LiveJour	**5.4M**	**79M**	**372**	Twitter	**41.6M**	**1.47G**	**2,488**	FriendST	68.3M	2.59G	304
Miscellaneous	FlickCTag	105.9K	2.3M	573	AmazMDS	334.8K	925.9K	6	CoActor	382.2K	33.1M	365
	AmazTWeb	403.3K	3.4M	10	Hollywood	**2.2M**	**229M**	**1,297**	DBpedia	3.97M	13.8M	20

Similarly, in Figs. 2e and 2f, we can clearly see that the results of the I/O costs are consistent with the results of the running time. Sampling is clearly the winner among all competitors, followed by SemiDegAppr, SemiDeg+, SemiStream, SemiCore, and SemiDeg. Both SemiDegAppr and SemiDeg+ use one order of magnitude less I/Os than SemiCore. For the memory overhead (reported in Figs. 2g and 2h), Sampling uses much more space than the other algorithms. All the other algorithms exhibit similar memory usages, because all those algorithms consume linear space. Note that Sampling is out of memory when running on the GSH dataset. These observations confirm our results shown in Section 4.

Disk-Based versus in-Memory Algorithms. Here we compare the time costs between SemiDeg+ and the state-of-the-art in-memory degeneracy compuation algorithm [48], called BucketCore, when the graph can fit in the main memory. BucketCore is an optimized in-memory core decomposition algorithm using a bucketing technique [48] which was shown to be faster than the traditional peeling-based core decomposition algorithm [32]. Note that for BucketCore, the input graph is stored in the main memory. However, for SemiDeg+, we only store the node information in the main memory and the edges of the input graph are stored in the disk, even if the whole graph can fit in the main memory. Fig. 3 shows the running time of SemiDeg+ and BucketCore on the five medium-sized graphs. As can be seen, SemiDeg+ is at least twice faster than BucketCore on these datasets. For example, on Arabic, SemiDeg+ takes 6.6 seconds, while BucketCore consumes 25.1 seconds to compute the degeneracy. The reason could be that SemiDeg+ directly computes the degeneracy based on an efficient binary-search procedure (with pruning optimization), while BucketCore needs to compute the core decomposition to derive the degeneracy which is typically more expensive than the binary-search procedure. These results indicate that the core-decomposition based algorithm is less efficient than the binary-search based algorithm for degeneracy computation.

Random versus Sequential I/O Costs. Recall that in SemiDeg+, the algorithm may incurs both random and



(a) Time cost (medium-sized graphs) (b) Time cost (massive graphs)

(c) Time cost (with preprocessing) (d) Time cost (with preprocessing)

(e) I/O cost (medium-sized graphs) (f) I/O cost (massive graphs)

(g) Memory cost (medium-sized graphs) (h) Memory cost (massive graphs)

Fig. 2. Results of various algorithms for degeneracy computation.

sequential I/O costs. When the algorithm starts to load the neighborhood of a node from the disk, the algorithm may incur a random I/O, because it needs to seek the position of that node's adjacency list in the disk. When loading an adjacency list into the main memory, the algorithm will take sequential I/Os, because an adjacency list may occupy several consecutive blocks in the disk. In this experiment, we study the number of random and sequential I/Os taken by SemiDeg+. Fig. 4 reports our results on the five massive graphs. We can see that the number of sequential I/Os is around 10 times larger than that of random I/Os on most datasets. Since the sequential I/Os are typically much cheaper than the random I/Os, thus our SemiDeg+ algorithm can be very efficient in practice, which are consistent with our previous results.

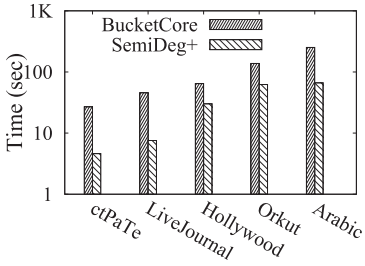


Fig. 3. Comparison SemiDeg+ with BucketCore.

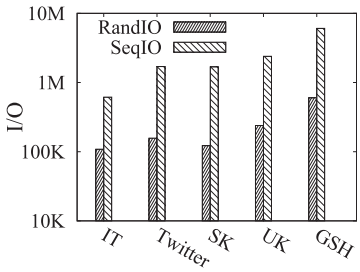
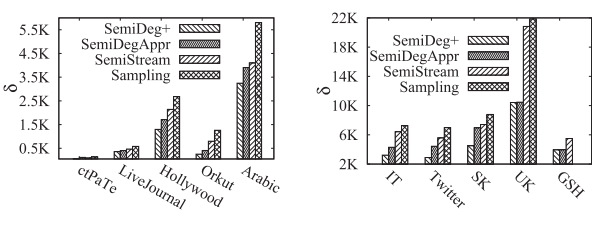


Fig. 4. Random versus sequential I/O costs for SemiDeg+.

Comparison of Approximation Algorithms. Here we evaluate the approximation preformation of SemiStream, Sampling, and SemiDegAppr. It should be noted that in SemiStream, a large parameter α will lead to better I/O performance, but it may degrade the approximation performance. In the previous experiment, we have already shown that both SemiDegAppr and SemiDeg+ are much more efficient than SemiStream even when $\alpha = 4$. Here we show that SemiDegAppr is also much better than SemiStream (with $\alpha = 4$) in terms of the approximation performance. The results are shown in Fig. 5. As can be seen, the degeneracy obtained by SemiDegAppr is near to optimal on many datasets, whereas both SemiStream and Sampling typically obtain a loose approximation of the degeneracy. For example, on the UK network, the exact degeneracy is 10,424 , while the degeneracies obtained by SemiDegAppr, SemiStream and Sampling are 10468, 20,828, and 21,860 respectively. These results suggest that SemiDegAppr is much better than SemiStream and Sampling for degeneracy computation on massive graphs in terms of approximation performance.

Scalability Testing. In this experiment, we show the scalability of SemiDegAppr and SemiDeg+ using Twitter and UK datasets. Similar results can also be observed on the other datasets. For both Twitter and UK, we generate four subgraphs by randomly sampling edges from 20 to 100 percent, and evaluate the time and I/O costs of our algorithms on these subgraphs.The results are shown in Fig. 6. As can be seen, both the running time and I/O costs of our algorithms increase as $|E|$ increases. The curves of both SemiDegAppr and SemiDeg+ are nearly linear, indicating that our algorithms scale very well in practice.

Results for Degeneracy Maintenance. In this experiment, we evaluate the performance of SemiDeg+ and SemiCore for degeneracy maintenance, since only SemiDeg+ and SemiCore can be used for degeneracy maintenance. We randomly delete and insert 1,000 edges in the graph for each test. The maintenance costs of each algorithm for edge deletion and edge insertion are the averaged results over 1,000 deletions and insertions respectively. The experimental results are shown in Fig. 7. From Fig. 7a, we can clearly see



(a) Medium-sized graphs (b) Massive graphs

Fig. 5. Approximation performance of different algorithms.

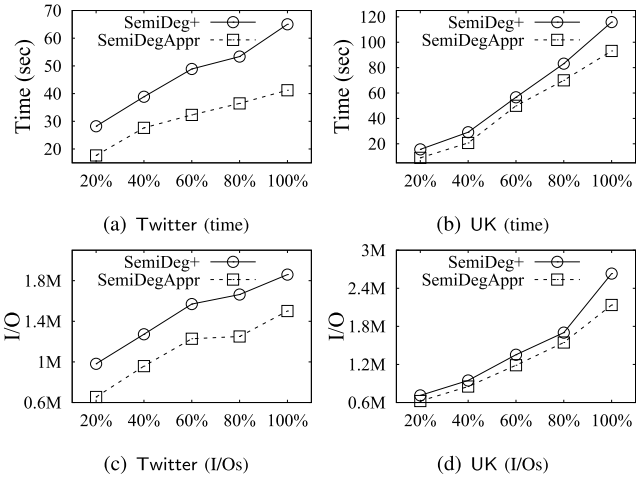


Fig. 6. Scalability testing (vary $|E|$).

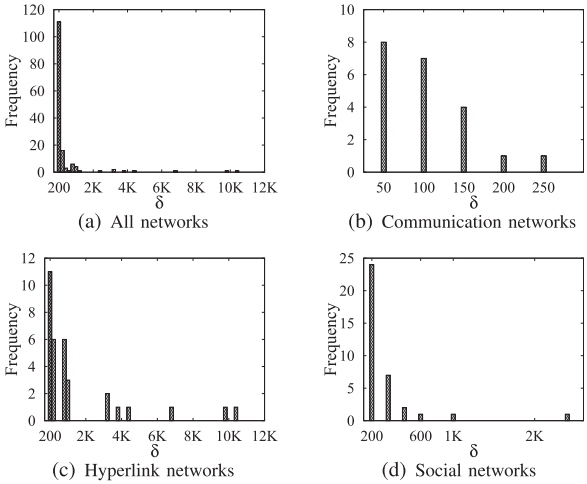


Fig. 8. Degeneracy distributions of real-world networks.

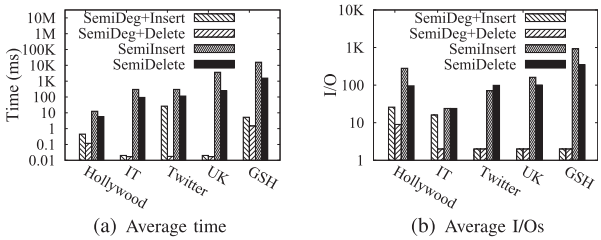


Fig. 7. Results of SemiDeg+ and SemiCore for degeneracy maintenance.

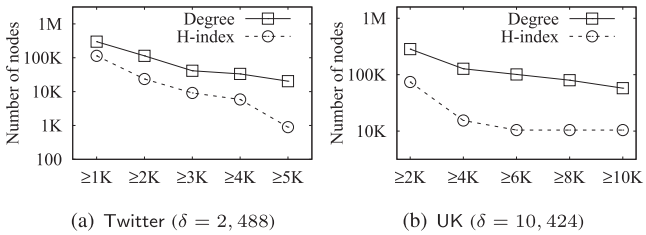


Fig. 9. Degree and h-index distributions of large-degeneracy graphs.

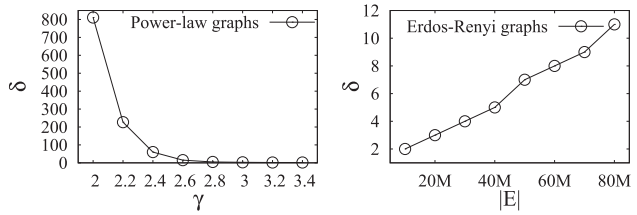
that SemiDeg+ is at least three orders of magnitude faster than SemiCore for handling an edge deletion on most networks. To process an edge insertion, SemiDeg+ is one order of magnitude faster than SemiCore on Hollywood and Twitter, and at least three orders of magnitude faster than SemiCore on IT, UK, and GSH. For example, on the Twitter dataset, SemiCore spends 115ms and 304ms to handle an edge deletion and insertion respectively, whereas SemiDeg+ takes only 0.02ms and 26ms to process an edge deletion and insertion respectively. This is because SemiDeg+ only needs to maintain the $c_{\max}$-core, while SemiCore has to maintain all the core numbers. Likewise, from Fig. 7b, we are able to derive similar results for the I/O costs of SemiDeg+ and SemiCore. These results confirm the theoretical analysis in Section 5.

6.3 Degeneracy of Different Networks

Degeneracy of Real-World Networks. In this experiment, we systematically evaluate the degeneracies of 150 real-world networks. The results are reported in Table 2. From Table 2, we can see that citation networks, collaboration networks, infrastructure networks, biology networks, software networks, lexical networks, computer networks, P2P networks, communication networks, and online contact networks have relatively small degeneracies. However, for some large social networks and hyperlink networks, the degeneracy can be very large. For example, the Twitter social network has a degeneracy 2,488, and the web graph UK has a degeneracy 10,424.

Fig. 8 depicts the degeneracy distributions of different types of networks. As can be seen, there are 111 networks that have a degeneracy smaller than 200, validating that many real-world networks indeed have small degeneracies. From Fig. 8c, we can observe that near one-half hyperlink networks have degeneracies larger than 800. Moreover, as reported in Table 2, all massive web graphs have very large degeneracies. From Fig. 8d, we can see that 80 percent social networks have small degeneracies ($\delta \leq 200$), and the remaining 20 percent social networks have relatively large degeneracies. These results indicate that the "small-degeneracy" assumption made in many existing work [2], [17], [22], [29], [35] might be excessively optimistic for web graphs and social networks.

Node distributions of large-degeneracy networks. Here we conduct an experiment to investigate why some real-world networks have large degeneracies. Specifically, we study the distributions of high-degree and high h-index nodes on the large-degeneracy networks. Fig. 9 shows the results on the Twitter and UK datasets. Similar results can also be obtained on the other large-degeneracy networks. As can be seen, both Twitter and UK contain a significant number of high-degree and high h-index nodes. These high-degree and high h-index nodes probably form a large dense subgraph which leads to the network having a large degeneracy. For example, on the UK dataset, there are 10,428 nodes that have h-index values no smaller than 10,000. By the definition of h-index, those high h-index nodes very likely form a dense subgraph, thus resulting in a large degeneracy value of UK.*Degeneracy of Random Graphs.* In this experiment, we evaluate the degeneracies of random graphs. We generate two sets of random graphs (with 10-million nodes): the power-law random graphs and the classic Erdos-Renyi (ER) random graphs. For the power-law random graphs, we vary the power-law degree exponent γ from 2 to 3.4, because most real-world power-law networks fall into this range [49]. For the ER graphs, we vary the number of edges from 10 million to 80 million. The results are shown in Fig. 10. From Fig. 10a, we can see that the



(a) Power-law graphs (vary γ) (b) ER graphs (vary $|E|$)

Fig. 10. Degeneracy of random graphs ($|V| = 10M$).

TABLE 3
The Degeneracy k-Core Community

Dataset	Degeneracy	$\lvert V_{C_{\max}}\rvert$	$\lvert E_{C_{\max}}\rvert$	Density
IT	3,224	3,240	5,247,052	0.9999
Twitter	2,488	3,192	4,585,552	0.9004
SK	4,510	4,514	10,185,835	0.9999
UK	10,424	10,427	54,355,948	0.9999

degeneracy of the power-law graph decreases with an increasing γ. Moreover, the degeneracy of the power-law graph is very small if $\gamma > 2.2$. These results further confirm that most real-world graphs have small degeneracies. On the other hand, the degeneracy of the ER graph increases as $|E|$ grows. This is because the density of the graph increases with increasing $|E|$, which may give rise to large k-cores [50], and therefore the degeneracy may increase.

Application for Community Detection. Note that our algorithms can also output the k-core with the largest k, termed as the degeneracy k-core. Here we show that the degeneracy k-core is often a very dense subgraph in real-life graphs, indicating that it can be used to detect communities in real-life graphs. Table 3 shows the detailed statistical information of the degeneracy k-cores on four massive graphs. As can be seen, the number of nodes of the degeneracy k-core is close to the degeneracy number and its density is near to 1, suggesting that the degeneracy k-core is very close to a clique. These results indicate that the proposed I/O-efficient algorithms can be applied to identify densely-connected communities in massive graphs.

7 Conclusion

In this paper, we propose a novel I/O-efficient algorithm using $O(n)$ memory to compute the degeneracy of massive graphs. We also devise an I/O-efficient degeneracy maintenance algorithm for dynamic graphs. Based on our algorithms, we perform a comprehensive evaluation of the degeneracy over 150 real-world graphs. The results suggest that most real-world graphs have small degeneracies, except for some large social networks and web graphs, in which the degeneracy can be up to several thousands. The experimental results also demonstrate that the proposed algorithms are substantially faster than the state-of-the-art algorithms for degeneracy computation and maintenance.

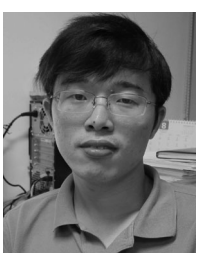
Acknowledgments

This work was partially supported by (i) NSFC Grants 61772346, U1809206, 61732003; (ii) National Key R&D Program of China 2018YFB1004402; (iii) Beijing Institute of Technology Research Fund Program for Young Scholars; (iv) Research Grants Council of the Hong Kong SAR, China No. 14202919 and 14203618; (v) ARC ARC FT200100787.

References

[1] D. Eppstein and D. Strash, "Listing all maximal cliques in large sparse real-world graphs," in *Proc. 10th Int. Symp. Exp. Algorithms*, 2011, pp. 364–375.
[2] D. Eppstein, M. Löffler, and D. Strash, "Listing all maximal cliques in large sparse real-world graphs," *ACM J. Exp. Algorithmics*, vol. 18, 2013, pp. 1–21.
[3] A. V. Goldberg, "Finding a maximum density subgraph," Univ. California at Berkeley, Berkeley, CA, Rep. no. 94720, 1984.
[4] Y. Dourisboure, F. Geraci, and M. Pellegrini, "Extraction and classification of dense communities in the web," in *Proc. 16th Int. Conf. World Wide Web*, 2007, pp. 461–470.
[5] Y. Dourisboure, F. Geraci, and M. Pellegrini, "Extraction and classification of dense implicit communities in the web graph," *ACM Trans. Web*, vol. 3, no. 2, pp. 7:1–7:36, 2009.
[6] L. Qin, R. Li, L. Chang, and C. Zhang, "Locally densest subgraph discovery," in *Proc. 21th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2015, pp. 965–974.
[7] N. Tatti and A. Gionis, "Density-friendly graph decomposition," in *Proc. 24th Int. Conf. World Wide Web*, 2015, pp. 1089–1099.
[8] Z. Li *et al.*, "Discovering hierarchical subgraphs of k-core-truss," *Data Sci. Eng.*, vol. 3, no. 2, pp. 136–149, 2018.
[9] G. Buehrer and K. Chellapilla, "A scalable pattern mining approach to web graph compression with communities," in *Proc. Int. Conf. Web Search Data Mining*, 2008, pp. 95–106.
[10] B. Saha, A. Hoch, S. Khuller, L. Raschid, and X. Zhang, "Dense subgraphs with restrictions and applications to gene annotation graphs," in *Proc. 14th Annu. Int. Conf. Res. Comput. Mol. Biol.*, 2010, pp. 456–472.
[11] D. Gibson, R. Kumar, and A. Tomkins, "Discovering large dense subgraphs in massive graphs," in *Proc. 31st Int. Conf. Very Large Data Bases*, 2005, pp. 721–732.
[12] M. Farach-Colton and M. Tsai, "Computing the degeneracy of large graphs," in *Proc. LATIN 11th Latin Amer. Symp. Theor. Inform.*, 2014, pp. 250–260.
[13] M. T. Goodrich and P. Pszona, "External-memory network analysis algorithms for naturally sparse graphs," in *Proc. 19th Eur. Conf. Algorithms*, 2011, pp. 664–676.
[14] M. Farach-Colton and M. Tsai, "Tight approximations of degeneracy in large graphs," in *Proc. LATIN 12th Latin Amer. Symp. Theor. Inform.*, 2016, pp. 429–440.
[15] C. S. J. A. Nash-Williams, "Decomposition of finite graphs into forests," *J. London Math. Soc.*, vol. 39, no. 1, pp. 12–12, 1964.
[16] B. Bollobas, *Extremal Graph Theory*. New York, NY, USA: Dover Publications, 2004.
[17] X. Hu, Y. Tao, and C.-W. Chung, "Massive graph triangulation," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 325–336.
[18] N. Chiba and T. Nishizeki, "Arboricity and subgraph listing algorithms," *SIAM J. Comput.*, vol. 14, no. 1, pp. 210–223, 1985.
[19] J. Wang and J. Cheng, "Truss decomposition in massive networks," *Proc. VLDB Endowment*, vol. 5, no. 9, pp. 812–823, 2012.
[20] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu, "Querying k-truss community in large and dynamic graphs," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2014, pp. 1311–1322.
[21] L. Chang, W. Li, X. Lin, L. Qin, and W. Zhang, "pSCAN: Fast and exact structural graph clustering," in *Proc. IEEE 32nd Int. Conf. Data Eng.*, 2016, pp. 253–264.
[22] R. Li, L. Qin, J. X. Yu, and R. Mao, "Influential community search in large networks," *Proc. VLDB Endowment*, vol. 8, no. 5, pp. 509–520, 2015.
[23] R. Li, L. Qin, J. X. Yu, and R. Mao, "Finding influential communities in massive networks," *VLDB J.*, vol. 26, no. 6, pp. 751–776, 2017.
[24] L. Chang, C. Zhang, X. Lin, and L. Qin, "Scalable top-k structural diversity search," in *Proc. IEEE 33rd Int. Conf. Data Eng.*, 2017, pp. 95–98.
[25] H. N. Gabow and H. H. Westermann, "Forests, frames, and games: Algorithms for matroid sums and applications," *Algorithmica*, vol. 7, no. 5&6, pp. 465–497, 1992.
[26] R. G. Downey and M. Fellows, *Parameterized Complexity*, Berlin, Germany: Springer, 1999.
[27] P. A. Golovach and Y. Villanger, "Parameterized complexity for domination problems on degenerate graphs," in *Proc. Int. Workshop Graph-Theoretic Concepts Comput. Sci.*, 2008, pp. 195–205.

[28] N. Alon and S. Gutner, "Linear time algorithms for finding a dominating set of fixed size in degenerated graphs," *Algorithmica*, vol. 54, no. 4, pp. 544–556, 2009.
[29] C. Lenzen and R. Wattenhofer, "Minimum dominating set approximation in graphs of bounded arboricity," in *Proc. Int. Symp. Distrib. Comput.*, 2010, pp. 510–524.
[30] N. Alon, R. Yuster, and U. Zwick, "Finding and counting given length cycles," *Algorithmica*, vol. 17, no. 3, pp. 209–223, 1997.
[31] D. W. Matula and L. L. Beck, "Smallest-last ordering and clustering and graph coloring algorithms," *J. ACM*, vol. 30, no. 3, pp. 417–427, 1983.
[32] V. Batagelj and M. Zaversnik, "An O(m) algorithm for cores decomposition of networks," *CoRR*, Tech. Rep. 0310049, vol. cs.DS/0310049, 2003.
[33] D. Wen, L. Qin, Y. Zhang, X. Lin, and J. X. Yu, "I/O efficient core graph decomposition at web scale," in *Proc. IEEE Int. Conf. Data Eng.*, 2016, pp. 133–144.
[34] B. Bahmani, R. Kumar, and S. Vassilvitskii, "Densest subgraph in streaming and MapReduce," *Proc. VLDB Endowment*, vol. 5, no. 5, pp. 454–465, 2012.
[35] M. C. Lin, F. J. Soulignac, and J. L. Szwarcfiter, "Arboricity, h-index, and dynamic algorithms," *Theor. Comput. Sci.*, vol. 426, pp. 75–90, 2012.
[36] D. R. Lick and A. T. White, "k-degenerate graphs," *Can. J. Math.*, vol. XXII, no. 5, pp. 1082–1096, 1970.
[37] S. B. Seidman, "Network structure and minimum degree," *Soc. Netw.*, vol. 5, no. 3, pp. 269–287, 1983.
[38] A. Aggarwal and J. S. Vitter, "The input/output complexity of sorting and related problems," *Commun. ACM*, vol. 31, no. 9, pp. 1116–1127, 1988.
[39] J. Cheng, Y. Ke, S. Chu, and M. T. Özsu, "Efficient core decomposition in massive networks," in *Proc. IEEE 27th Int. Conf. Data Eng.*, 2011, pp. 51–62.
[40] A. Montresor, F. D. Pellegrini, and D. Miorandi, "Distributed k-core decomposition," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 2, pp. 288–300, Feb. 2013.
[41] J. E. Hirsch, "An index to quantify an individual's scientific research output," *Proc. Nat. Acad. Sci. United States America*, vol. 102, no. 46, pp. 16 569–16 572, 2005.
[42] D. Eppstein and E. S. Spiro, "The h-index of a graph and its application to dynamic subgraph statistics," *J. Graph. Algorithms Appl.*, vol. 16, no. 2, pp. 543–567, 2012.
[43] J. Cheng, Y. Ke, A. W.-C. Fu, J. X. Yu, and L. Zhu, "Finding maximal cliques in massive networks," *ACM Trans. Database Syst.*, vol. 36, no. 4, pp. 21:1–21:34, 2011.
[44] L. Lü, T. Zhou, Q.-M. Zhang, and H. E. Stanley, "The h-index of a network node and its relation to degree and coreness," *Nature Commun.*, vol. 7, no. 10168, pp. 1–7, 2016.
[45] J. Jiang, M. Mitzenmacher, and J. Thaler, "Parallel peeling algorithms," in *Proc. Symp. Parallelism Algorithms Architectures*, 2014, pp. 319–330.
[46] R. Li, J. X. Yu, and R. Mao, "Efficient core maintenance in large dynamic graphs," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 10, pp. 2453–2465, Oct. 2014.
[47] A. E. Sariyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and Ü. V. Çatalyürek, "Streaming algorithms for k-core decomposition," *Proc. VLDB Endowment*, vol. 6, no. 6, pp. 433–444, 2013.
[48] L. Dhulipala, G. E. Blelloch, and J. Shun, "Julienne: A framework for parallel graph algorithms using work-efficient bucketing," in *Proc. 29th ACM Symp. Parallelism Algorithms Architectures*, 2017, pp. 293–304.
[49] A.-L. Barabasi and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, pp. 509–512, 1999.
[50] S. N. Dorogovtsev, A. V. Goltsev, and J. F. F. Mendes, "k-core organization of complex networks," *Phys. Rev. Lett.*, vol. 96, no. 4, 2006, Art. no. 040601.



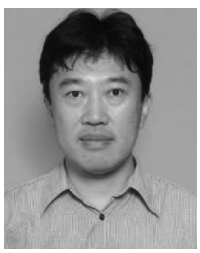
Rong-Hua Li received the PhD degree from the Chinese University of Hong Kong, in 2013. He is currently a professor with the Beijing Institute of Technology, Beijing, China. His research interests include graph data management and mining, social network analysis, graph computation systems, and graph-based machine learning.

Qiushuo Song received the bachelor's degree from Shantou University, Shantou, China, in 2016, and the master's degree from Shenzhen University, Shenzhen, China. His current research interests include graph data management, graph mining, and social network analysis.

Xiaokui Xiao received the PhD degree in computer science from the Chinese University of Hong Kong, in 2008. He is currently an associate professor with the National University of Singapore (NUS), Singapore. Before joining NUS in 2018, he was a associate professor with Nanyang Technological University (NTU). His research interests include data privacy, spatial databases, graph databases, and parallel computing.

Lu Qin received the bachelor's degree from the Department of Computer Science and Technology, 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 now a associated professor with the Centre of Quantum Computation and Intelligent Systems (QCIS), University of Technology, Sydney (UTS). His research interests include parallel big graph processing, I/O efficient algorithms on massive graphs, and keyword search in relational database.

Guoren Wang received the BSc, MSc, and PhD degrees from the Department of Computer Science, Northeastern University, China, in 1988, 1991, and 1996, respectively. Currently, he is a professor with the Department of Computer Science, Northeastern University, China. His research interests include XML data management, query processing and optimization, bioinformatics, high dimensional indexing, parallel database systems, and cloud data management.



Jeffery Xu Yu received the BE, ME, and PhD degrees in computer science from the University of Tsukuba, Japan, in 1985, 1987, and 1990, respectively. He has held teaching positions at the Institute of Information Sciences and Electronics, University of Tsukuba, and with the Department of Computer Science, Australian National University, Australia. Currently, he is a professor with the Department of Systems Engineering and Engineering Management, Chinese University of Hong Kong, Hong Kong. His current research interests include graph database, graph mining, keyword search in relational databases, and social network analysis.

Rui Mao received the PhD degree in computer science from the University of Texas at Austin, in 2007. He is currently a professor with Shenzhen University. His research interests include big data analysis and management, content-based similarity query of multimedia and biological data, data mining, and machine learning.