Computing (1 + ϵ)-Approximate Degeneracy in Sublinear Time

Valerie King, **Alex Thomo**, **Quinton Yong**
University of Victoria, Victoria, Canada
{val, thomo, quintonyong}@uvic.ca

Abstract

The problem of finding the degeneracy of a graph is a subproblem of the k-core decomposition problem. In this paper, we present a $(1+\epsilon)$-approximate solution to the degeneracy problem which runs in $O(n \log n)$ time, sublinear in the input size for dense graphs, by sampling a small number of neighbors adjacent to high degree nodes. This improves upon the previous work on sublinear approximate degeneracy, which implies a $(4 + \epsilon)$-approximate $\tilde{O}(n)$ solution. Our algorithm can be extended to an approximate $O(n \log n)$ time solution to the k-core decomposition problem. We also explore the use of our approximate algorithm as a technique for speeding up exact degeneracy computation. We prove theoretical guarantees of our algorithm and provide optimizations, which improve the running time of our algorithm in practice. Experiments on massive real-world web graphs show that our algorithm performs significantly faster than previous methods for computing degeneracy.

1 Introduction

The degeneracy of an undirected graph G, denoted δ, is the smallest δ such that every induced subgraph of G has a node with degree at most δ. The problem of finding the degeneracy of a graph is a subproblem of the k-core decomposition problem, where a k-core is a maximal connected subgraph in which all nodes have degree at least k, and k-core decomposition problem is to find all k-cores of a given graph G. The degeneracy of G is the maximum k for which G contains a (non-empty) k-core.

The degeneracy of a graph is used for many applications in graph analysis and graph theory including measuring the sparcity of a graph, approximating dominating sets [Lenzen and Wattenhofer, 2010; Dvorak, 2013], approximating the arboricity of a graph within a constant factor [Eden *et al.*, 2022], and maximal clique enumeration [Eppstein and Strash, 2011]. It is also known that the degeneracy is at least half the maximum density of any subgraph [Farach-Colton and Tsai, 2014]. The k-core decomposition and degeneracy of graphs also have real world applications including analyzing social networks [Bhawalkar *et al.*, 2012], graph visualization [Alvarez-Hamelin *et al.*, 2005], and describing protein functions in protein-protein interaction networks [Altaf-Ul-Amin *et al.*, 2006]. Degeneracy is also used as a feature in machine learning applications such as network similarity analysis [Nikolentzos *et al.*, 2018], graph representation learning for drug discovery and recommender systems [Brandeis *et al.*, 2020], and neural network initialization [Limnios *et al.*, 2021]. Thus, it is extremely beneficial to be able to efficiently compute the degeneracy on large graphs.

The k-core decomposition problem, and consequently the degeneracy problem, can be solved using a well known linear time "peeling" algorithm [Matula and Beck, 1983]. Starting with $k = 1$, the algorithm repeatedly removes all nodes with degree less than k and their incident edges, until the only remaining nodes have degree at least k and their connected components are k-cores. Then k is incremented, and the algorithm is repeated on the remaining subgraph. There has been much work done to improve the practical running time of k-core decomposition fully in-memory [Batagelj and Zaversnik, 2003], with fully external-memory [Cheng *et al.*, 2011], and with semi external-memory [Wen *et al.*, 2015]. There are also algorithms for approximating k-core in dynamic, streaming, and sketching settings [Li *et al.*, 2014; Saríyüce *et al.*, 2013; Esfandiari *et al.*, 2018]. A sublinear approximate algorithm for degeneracy is the algorithm presented in [Bhattacharya *et al.*, 2015] for computing maximum density of any subgraph. It runs in $\tilde{O}(n)$ time and implies a $(4 + \epsilon)$-approximate algorithm for degeneracy.

While k-core algorithms can be used to compute degeneracy, they may incur unnecessary costs and space for processing cores for small k. There are algorithms which directly compute the degeneracy of the graph. The SemiDeg+ algorithm from [Li *et al.*, 2022] is the state-of-the-art in exact degeneracy computation. In a semi-streaming model, [Farach-Colton and Tsai, 2014], [Goodrich and Pszona, 2011], and [Bahmani *et al.*, 2012] propose $(2 + \epsilon)$-approximate solutions and [Farach-Colton and Tsai, 2016] proposes a $(1 + \epsilon)$-approximate solution. These require each edge to be scanned, however [Farach-Colton and Tsai, 2016] can be modified to run more efficiently for the adjacency list model.

In this work, we present a $(1 + \epsilon)$-approximate $O(\frac{n \log n}{\epsilon^3})$ time solution to the degeneracy problem where n is the number of nodes in the graph. Hence, our algorithm is sublinear in the number of edges for dense graphs. We avoid the overhead of full k-core decomposition and directly compute degeneracy of the given graph G by sampling the neighbors of nodes starting with those with the highest degree above a certain threshold l. We then determine if the subgraph induced by those nodes contains an l-core by repeatedly eliminating nodes which did not sample a sufficient number of other high degree nodes, similar to the peeling algorithm. If it does, we output l as the (approximate) degeneracy of G. We prove the theoretical guarantees of our algorithm and we also propose optimizations which improve its practical running time. Our algorithm can also be extended to compute exact degeneracy and to a sublinear time algorithm for k-core decomposition.

We compare our solution to the $(1+\epsilon)$-approximate algorithm of [Farach-Colton and Tsai, 2016] and the SemiDeg+ algorithm from [Li *et al.*, 2022] on massive real-world webgraphs and show that our algorithm is faster than both on all datasets. We also show that our algorithm can sample asymptotically less than the algorithm from [Farach-Colton and Tsai, 2016] modified for the adjacency list model. Our main theoretical result is the following:

Theorem 1. *There is an algorithm for k-core decomposition which runs in time $O(\frac{n \log n}{\epsilon^3})$ and gives a $(1+\epsilon)$ factor approximation for the core values of a graph with n nodes and m edges in the adjacency-list model, with high probability.*

2 Preliminaries

2.1 Definitions and Notation

Let $G = (V, E)$ be an undirected graph with nodes V and edges E, where $|V| = n$ and $|E| = m$. We denote the degree of a vertex $v \in V$ in G as $\deg(v)$. A subgraph of G induced by a subset of nodes $V' \subseteq V$ is defined as $G_{V'} = (V', E')$ where $E' = \{(u, v) \mid u, v \in V', (u, v) \in E\}$. We denote the degree of a vertex v in $G_{V'}$ as $d_{G_{V'}}(v)$.

For a graph G and an integer k, a k-core of G is the maximal induced connected subgraph of G such that every node in the subgraph has degree at least k [Seidman, 1983]. The core number of a node $v \in V$ is the largest k for each there is a k-core that contains v. The degeneracy δ of a graph G is defined as the smallest integer such that every non-empty induced subgraph has a vertex with degree at most δ [Lick and White, 1970]. The degeneracy of G is also equal to the largest k for which G contains a non-empty k-core [Eppstein and Strash, 2011].

2.2 Problem Definition

Our goal is to design an efficient algorithm for computing a $(1+\epsilon)$ approximation of the degeneracy δ of a given graph G for $\epsilon \in (0, 1]$, where ϵ is an input parameter, with high probability. The output of the algorithm $\hat{\delta}$ is a $(1+\epsilon)$ approximation if $\delta/(1+\epsilon) \leq \hat{\delta} \leq \delta \cdot (1+\epsilon)$. We define "with high probability" ("w.h.p.") as occurring with probability at least $1 - \frac{O(1)}{n^c}$, where c is a constant given as an input parameter.

2.3 Graph Input Model

The graph input model we use is the adjacency list model, where each node $v \in V$ has an adjacency list of the neighbors of v, arbitrarily ordered. This is a standard graph representation used in sublinear time algorithms [Bhattacharya *et al.*, 2015; Chazelle *et al.*, 2005]. There are two types of queries to access graph information in this model:

1) degree queries to get the degree of any node $v \in V$
2) neighbor queries to get the i^{th} neighbor of any node $v \in V$, which is the i^{th} element in v's adjacency list

3 Related Work

Matula and Beck [1983] presented the linear time peeling algorithm for k-core decomposition. Although it is superseded in practice by heuristic methods in other work and presents unnecessary overhead for degeneracy computation (as do other algorithms which enumerate all k-cores), it is an efficient method for core computation especially for small graphs and graphs with small degeneracy.

The work by Esfandiari et al. [2018] and Colton et al. [2016] present $(1+\epsilon)$-approximate solutions for k-core decomposition and degeneracy computation respectively. The goal of both algorithms is to minimize memory usage in a semi-streaming model, which make them inefficient in terms of running time in the adjacency list model due to the requirement of making a random choice for every edge in the graph, especially with large graphs. However, the algorithm from [Farach-Colton and Tsai, 2016] can be modified to run more efficiently in the adjacency list model.

Bhattacharya et al. [2015] proposes a $(2+\epsilon)$-approximate solution to the densest subgraph problem in sublinear time, which implies a $(4+\epsilon)$-approximate sublinear algorithm for degeneracy, and uses it also uses the adjacency list model for their graph input representation. The SemiDeg+ algorithm [Li *et al.*, 2022] is the state-of-the-art in practical exact degeneracy computation which uses binary search within known bounds for degeneracy while maintaining an upper bound of core values on nodes to speed up verification of if a core exists. We use it as a comparison baseline for the experiments on the running times of our approximate algorithm.

Ghaffari et al. [2019] improves the work of [Esfandiari *et al.*, 2018] in a parallel computing environment by reducing the number of parallel computation rounds required, while maintaining a $(1+\epsilon)$ approximation ratio, and uses a vertex sampling technique instead of edge sampling. Eden et al. [2022] present an $O(\log^2 n)$-approximate sublinear time algorithm for finding the arboricity of a graph in the adjacency list model which uses a technique similar to [Bhattacharya *et al.*, 2015].

4 Degeneracy Algorithm

4.1 High Level Description

The key insight behind the algorithm is that the maximum core of the graph will contain high degree nodes which are connected to many other high degree nodes. So, we can efficiently compute the approximate degeneracy of the graph by looking at the subgraph H induced by the high degree nodes. Initially, the nodes in V are put into H if their degree is above a certain threshold l, where l is what we approximate as the degeneracy of the graph, and we decrease l in each iteration when there is a high probability that H does not contains an l-core (within an approximation factor of $(1+\epsilon)$). To efficiently check if there is a high probability that H does contain an l-core, we sample a small number of neighbors adjacent to each node in H. If a node $v \in H$ is in the l-core, then it is likely that it samples a sufficient fraction of its number of neighbors that are also in H. If a node does not sample a sufficient number of neighbors in H, then it is likely not part of an l-core, and we remove the node from H and check its neighbors and remove them from H if necessary (similar to the peeling algorithm). If no nodes remain unpeeled, then we lower the threshold and repeat the procedure. Otherwise, we conclude that H has an approximate core number of l, and we output l as the approximate degeneracy of G. In Section 5, we prove that the output is within a factor of $(1+\epsilon)$ of the true degeneracy of G with probability $\geq 1 - \frac{2}{n^c}$.

4.2 Algorithm Details

The approximate algorithm for computing degeneracy is illustrated in Algorithm 1. In each iteration of the while-loop at Line 3, we partition the nodes in V into H and L, where H contains the nodes which have a degree greater than or equal to a certain threshold l and $L = V \setminus H$ (Line 4). Initially, $l = \frac{n}{(1+\epsilon_1)}$, where $\epsilon_1 = \epsilon/3$, and we decrease l by a factor of $(1+\epsilon_1)$ (Lines 1 and 27). Then, we sample a small number of neighbors for each node $v \in H$ (Line 10). We sample $k(v)$ neighbors u of v with replacement for every node $v \in H$, where $k(v) = \lceil p \deg(v) \rceil$. Initially $p = \frac{2((1+c)\ln(n)+\ln(\log_{1+\epsilon_1} n))(1+\epsilon_1)^2}{\epsilon_1^2 n}$ and we increase p by a factor of $(1+\epsilon_1)$ in each iteration (Lines 1 and 28).

Next, we check if H contains an l-core. This is done by a process similar to peeling, in which we repeatedly peel nodes from H which did not sample a sufficient number of neighbors in H. In particular, each node $v \in H$ is assigned a value $t(v)$ which is initially set to the number of neighbors sampled by v, $k(v)$, and represents the degree of v in H (Line 8). Then, since we sampled a $k(v)/\deg(v)$ fraction of neighbors, we expect that at least $l \cdot k(v)/\deg(v)$ edges incident to other nodes in H were sampled if the current nodes in H is an l-core. So, if at any point $t(v)$ for a node $v \in V$ falls below $\frac{l \cdot k(v)}{\deg(v)}$, then v is peeled from H to L (Lines 13 and 22).

In the sampling process (Lines 9 - 17), whenever a node $v \in H$ samples a neighbor $u \in L$, we decrement $t(v)$ (Line 12). Otherwise, if it samples a neighbor $u \in H$, we must remember that u was sampled by v so if u is peeled from H later, we also decrement $t(v)$. Each node in $u \in H$ maintains an initially empty list (with possible repeats) $Sampled(u)$ of nodes which sampled them. So, if $v \in H$ samples a neighbor $u \in H$, we add v to $Sampled(u)$ (Line 17). Then, if a node in $u \in H$ is peeled due to $t(u)$ falling below $\frac{l \cdot k(u)}{deg(u)}$, we add u to an initially empty queue Q (Line 15).

After the sampling process is finished, we remove each node u from Q, iterate through each node $v \in Sampled(u)$, and decrement $t(v)$ if v is still in H (Line 22). If $t(v)$ for a node $v \in H$ falls to less than $\frac{l \cdot k(v)}{\deg(v)}$, then v is moved to L and then is also added to Q (Lines 24 and 25). If there are no remaining nodes in H, that means each node in H did not sample a sufficient number of other nodes in H, and it is likely that there is no l-core in G. So, we continue to the next iteration where we repeat the process with $l = l/(1+\epsilon_1)$, and $p = p \cdot (1+\epsilon_1)$, and we also set the sampled lists to empty (Lines 27 - 29). Otherwise, if there are nodes remaining in H, then we expect that H is at least an l-core and l is the highest threshold for which we found a core thus far. Therefore, we return l as an approximation to the degeneracy of G (Line 31). In the case where $p \geq 1$ before l is found, we simply compute δ by running the peeling algorithm and return δ.

Algorithm 1 ApproximateDegeneracy(G, ϵ, c)

Input: Graph $G = (V, E)$, error factor $\epsilon \in (0, 1]$, constant c
Output: Approximate degeneracy of G within a factor of $(1+\epsilon)$ with probability $\geq 1 - \frac{2}{n^c}$

```
1:  $\epsilon_1 = \epsilon/3, l = \frac{n}{(1+\epsilon_1)}, p = \frac{2((1+c)\ln(n)+\ln(\log_{1+\epsilon_1} n))(1+\epsilon_1)^2}{\epsilon_1^2 n}$
2:  $Sampled(v) \leftarrow \emptyset$ for all $v \in V$
3:  while $p < 1$ do
4:      $H \leftarrow \{v \in V \mid \deg(v) \geq l\}, L \leftarrow V \setminus H$
5:      $Q \leftarrow \emptyset$
6:      for each node $v \in H$ do
7:          $k(v) \leftarrow \lceil p \deg(v) \rceil$
8:          $t(v) \leftarrow k(v)$
9:          for $i \in 1, ..., k(v)$ do
10:             pick a neighbor $u$ of $v$ independently at random
11:             if $u \in L$ then
12:                 $t(v) = t(v) - 1$
13:                 if $t(v) < \frac{l \cdot k(v)}{\deg(v)}$ then
14:                     $H \leftarrow H \setminus \{v\}, L \leftarrow L \cup \{v\}$
15:                     add $v$ to $Q$
16:             else
17:                 add $v$ to $Sampled(u)$
18:     while $Q \neq \emptyset$ do
19:         $u \leftarrow dequeue(Q)$
20:         for each node $v \in Sampled(u)$ do
21:             if $v \in H$ then
22:                 $t(v) = t(v) - 1$
23:                 if $t(v) < \frac{l \cdot k(v)}{\deg(v)}$ then
24:                     $H \leftarrow H \setminus \{v\}, L \leftarrow L \cup \{v\}$
25:                     add $v$ to $Q$
26:     if $H = \emptyset$ then
27:         $l \leftarrow l/(1+\epsilon_1)$
28:         $p \leftarrow p \cdot (1+\epsilon_1)$
29:         $Sampled(v) \leftarrow \emptyset$ for all $v \in V$
30:     else
31:         return $l$
32: compute $\delta$ by running the peeling algorithm
33: return $\delta$
```

5 Theoretical Guarantees

In this section, we will prove the probability of correctness and sublinear running time of Algorithm 1.

5.1 Correctness

The correctness depends on showing that our sampling of edges is effective, that is, for any node in H, w.h.p., $t(v)$, when appropriately scaled, gives a tight approximation of v's degree in H. The tightness relies on the application of Chernoff bounds, which can be used only because the edges incident to any one node v in any one iteration are sampled independently at random. Let (H, L) be a partition of the vertices of G. Initially $H = V$ and $L = \emptyset$. Let $d_H(v)$ denote the degree of a vertex $v \in H$ in the subgraph induced by the nodes in H. Lemma 1 enables us to regard each sample as an independent uniformly random coin flip, whose outcome is heads if the neighbor u incident to the sampled edge is in H; the probability of heads is $d_H(v)/\deg(v)$.

Lemma 1. *At any time during an iteration, the event that, in a single execution of Line 10, the neighbor picked by v is in H has probability $d_H(v)/\deg(v)$, and the set of events corresponding to the $k(v)$ executions of Line 10 are mutually independent.*

Proof. Consider H at a specific time. Let E_i be the event that v's i^{th} sample (Line 10) is in H. The event that any one neighbor is selected is an independent uniformly random event with probability with probability $1/\deg(v)$. These random choices are independent from the state of H at any time and they are independent of each other since they are picked with replacement. Thus, at any point in the algorithm, if we fix the nodes in H, the probability of any node v sampling a neighbor in H is exactly the number of its neighbors in H, $d_H(v)$, over its degree. $\square$

Lemma 2 states the value of $t(v)$ at any time for any $v \in H$.

Lemma 2. *At any time, for any node $v \in H$, $t(v)$ is the number of v's sampled neighbors in H.*

Proof. For each node $v \in H$, $t(v)$ is initialized to the number of samples, $k(v)$. Whenever v discovers that a sampled neighbor is not in H, either in Line 11 or Line 21, $t(v)$ is decremented. So, $t(v) = k(v) -$ the number of sampled neighbors in L. $\square$

Then, we have that Lemma 3 follows from Lemma 2.

Lemma 3. *The expected value of $t(v)$ for each node $v \in H$ is $E[t(v)] = \frac{d_H(v)}{\deg(v)}k(v)$.*

Now, we will prove that the bound $\frac{l \cdot k(v)}{\deg(v)}$ (Line 13 and 23), which we require for $t(v)$ for a node v to remain in H, gives us an approximate bound for the degree of v in H.

Lemma 4. *When a node is moved to L, for every node $v \in H$*

1) if $d_H(v) \geq l \cdot (1 + \delta_1)$, then $t(v) \geq \frac{l \cdot k(v)}{\deg(v)}$, and

2) if $d_H(v) < l/(1 + \delta_2)$, then $t(v) < \frac{l \cdot k(v)}{\deg(v)}$

w.h.p., where $\delta_1 = \epsilon_1$ and $\delta_2 = \frac{3}{2}\epsilon_1$ for $\epsilon_1 \leq 1$.

Proof. By Lemma 1, each sample can be regarded as an independent random coin flip which is heads when a sampled neighbor is in H. We will use the following Chernoff bounds from Theorem 2.1 in [Aamand *et al.*, 2021]. Let X be the sum of the number of heads and let $\mu = E[X]$. For $0 < \delta_1, \delta_2$

$$\Pr(X < (1 - \delta_1)\mu') \leq e^{\frac{-\delta_1^2 \mu'}{2}} \text{ for all } \mu' \leq \mu \tag{1}$$

and

$$\Pr(X \geq (1 + \delta_2)\mu') \leq e^{\frac{-\delta_2^2 \mu'}{2+\delta_2}} \text{ for all } \mu' \geq \mu \tag{2}$$

In our experiment, we have $k(v)$ coin flips where the probability of heads is $d_H(v)/\deg(v)$ and by Lemma 3, $E[t(v)] = \frac{d_H(v)}{\deg(v)}k(v) = \mu$.

To prove 1), when $d_H(v) \geq l \cdot (1 + \delta_1)$, let $\mu' = (1 + \delta_1) \cdot l \cdot k(v)/\deg(v)$. If $d_H(v) \geq l \cdot (1 + \delta_1)$, then $E[t(v)] \geq \mu'$. Now, we can bound the probability that $t(v) < l \cdot k(v)/\deg(v)$, which we can rewrite as $t(v) < (1 - \frac{\delta_1}{1+\delta_1})\mu'$. By the Chernoff bound in Equation 1, the probability that $t(v) < (1 - \frac{\delta_1}{1+\delta_1})\mu'$ is $\leq e^{\frac{-\delta_1^2 \mu'}{2(1+\delta_1)^2}} = e^{\frac{-\delta_1^2 \cdot l \cdot k(v)/\deg(v)}{2(1+\delta_1)}}$.

To prove 2), when $d_H(v) < l/(1 + \delta_2)$, let $\mu' = \frac{l}{(1+\delta_2)}\frac{k(v)}{\deg(v)} > \frac{d_H(v)}{\deg(v)}k(v) = \mu$. By the Chernoff bound in Equation 2, $\Pr(t(v)) \geq l \cdot k(v)/\deg(v)) = \Pr(t(v) \geq (1 + \delta_2)\mu') \leq e^{\frac{-\mu'\delta_2^2}{2+\delta_2}} = e^{\frac{-\delta_2^2 l \cdot k(v)/\deg(v)}{(1+\delta_2)(2+\delta_2)}}$.

Since $k(v) \geq p \cdot \deg(v)$, we have $l \cdot k(v)/\deg(v) \geq l \cdot p$. So, the probability that 1) fails to hold is $\leq e^{\frac{-\delta_1^2 l \cdot p}{2(1+\delta_1)}}$ and the probability that 2) fails to hold is $\leq e^{\frac{\delta_2^2 l \cdot p}{(1+\delta_2)(2+\delta_2)}}$. We let $\delta_2 = (3/2)\delta_1$ to get that bound on the probability of error 2) is $e^{\frac{-(9/4)\delta_1^2 l \cdot p}{2+(9/2)\delta_1+(9/4)\delta_1^2}} \leq e^{\frac{-\delta_1^2 l \cdot p}{2(1+\delta_1)}}$ when $\delta_1 \leq 1$. Recall that $\delta_1 = \epsilon_1$.

Then, we take the union bound of both error probabilities 1) and 2) over all n nodes. Since l is initialized to $n/(1 + \epsilon_1)$, and we decrease l by a factor of $(1 + \epsilon_1)$ in each iteration, the algorithm will take less than $\log_{1+\epsilon_1} n$ iterations. So, we multiply both errors 1) and 2) by $e^{\ln n + \ln \log_{1+\epsilon_1} n}$ for the union bound and over all possible iterations. Therefore, the probability each error is less than $1/n^c$ when $l \cdot p \geq \frac{2((1+c)\ln n + \ln \log_{1+\epsilon_1} n)(1+\epsilon_1)}{\epsilon_1^2}$ and the probability any error occurs is less than $2/n^c$ for any c. $\square$

Next, we show that in any iteration of Algorithm 1, $d_H(v)$ for any node $v \in H$ satisfies the following conditions w.h.p.

Lemma 5. *In any iteration of Algorithm 1,*

1) every node $v \in H$ is moved to L when $d_H(v) < l/(1 + (3/2)\epsilon_1)$

2) no node $v \in H$ is moved to L when $d_H(v) \geq l \cdot (1 + \epsilon_1)$

w.h.p.

Proof. By Lemma 4, w.h.p., no node $v \in V$ with $d_H(v) > l \cdot (1 + \epsilon_1)$ has $t(v) < l \cdot k(v)/\deg(v)$, so v is not moved to L. Also, every node $v \in H$ with $d_H(v) < l/(1 + (3/2)\epsilon_1)$ has $t(v) < l \cdot k(v)/\deg(v)$, so v is moved to L. $\square$

We can now relate $d_H(v)$ to the core values of the nodes in L and the nodes in H.

Lemma 6. *Let $d_1 = l \cdot (1 + \epsilon_1)$ and $d_2 = l/(1 + (3/2)\epsilon_1)$. At Line 26 of Algorithm 1, w.h.p., the nodes in H have core values at least d_2 and the nodes in L have core values less than d_1.*

Proof. By condition 1 of Lemma 5, every node in H has $\geq d_2$ neighbors in H. So, every node in H must have a core value of at least d_2.

Suppose for a contradiction that a node is moved to L with a core value $\geq d_1$. Then there must be a first node u for which this happens. Since the core value of u is $\geq d_1$ by the assumption, u must have $\geq d_1$ neighbors with core values at least d_1. Since u is the first such node to be moved to L with core value $\geq d_1$, none of u's neighbors which are in the d_1-core have been moved to L. Hence, $d_H(u) \geq d_1$ and u cannot be moved to L by condition 2 of Lemma 5. ☐

We can now prove the bounds on the approximate degeneracy output by Algorithm 1.

Lemma 7. *Let l be the output returned by the algorithm in line 31 in the case a value is returned. W.h.p., for each $v \in V$, $c(v) < l\cdot(1+\epsilon_1)^2$ and for all $v \in H$, $c(v) \geq l/(1+(3/2)\epsilon_1)$. In particular, $\delta/(1+\epsilon_1)^2 < l \leq \delta(1+(3/2)\epsilon_1)$.*

Proof. In the iteration when the threshold was $l\cdot(1+\epsilon)$, all nodes in H were moved to L. By Lemma 5, at the end of the algorithm, when the threshold is l, $d_H(v)$ for each node $v \in H$ satisfies $d_H(v) \geq l/(1+(3/2)\epsilon_1)$ and since v was moved to L in the previous iteration, $d_H(v) < l\cdot(1+\epsilon_1)^2$. Let $c(v)$ denote the core number of a node v. Then by Lemma 6, for each $v \in H$ we have $c(v) < l\cdot(1+\epsilon_1)^2$ and $c(v) \geq l/(1+(3/2)\epsilon_1)$, which gives us the bounds $c(v)/(1+\epsilon_1)^2 < l \leq c(v)(1+(3/2)\epsilon_1)$. Since l is the highest threshold for which not all nodes in H were moved to L, it follows that $\delta/(1+\epsilon_1)^2 < l \leq \delta(1+(3/2)\epsilon_1)$. ☐

We observe that if we set $\epsilon_1 = \epsilon/3$, we get $(1+\epsilon/3)^2 = 1+2\epsilon/3+\epsilon^2/9$ which is less than $1+\epsilon$ when $\epsilon \leq 1$. Also, if no l is returned and $p \geq 1$, we run the peeling algorithm and return δ. Thus, Algorithm 1 gives a $(1+\epsilon)$ approximation.

Lemma 8. *Given a graph G, Algorithm 1 outputs a $(1+\epsilon)$ approximation to the degeneracy δ of G.*

5.2 Running Time

Let j be the number of iterations (of the while-loop at Line 3) which Algorithm 1 takes to compute the approximate degeneracy of an input graph G. When the output $l = n/(1+\epsilon_1)^j$, then $j = \log_{1+\epsilon_1}(n/l)$. Consider the cost of sampling the neighbors of any node $v \in H$. Let $k_i(v)$ and p_i be the number of neighbors sampled and the value of p, respectively, in any iteration i of the algorithm. The number of neighbors sampled by v over the algorithm is at most $\log_{1+\epsilon_1}(n/l) + \frac{2((1+c)\ln(n)+\ln(\log_{1+\epsilon_1} n))(1+\epsilon_1)^3}{\epsilon_1^3 l}\deg(v)$.[1]

When sampling neighbors, every edge (u,v) may be examined when sampled in line 10 by u or v or both. We can attribute the cost of sampling the endpoints of an edge to one of its endpoints and take twice the cost to get an upper bound on the number of samples made.

First, we consider the nodes $v \in H$ which remain in H by the end of iteration j. In iteration $j-1$, v was moved to L. For each edge (u,v), either u was moved to L before v, in which case we attribute the edge in iteration j to u, otherwise we attribute the edge to v. The number of remaining edges incident to v, which are incident to nodes in H when v was moved to L, is $d_H(v)$ at the time of the move (in iteration $j-1$). By Lemma 5, $d_H(v) < l\cdot(1+\epsilon_1)^2$ w.h.p. Thus, the total number of edges attributed to v is less than $l\cdot(1+\epsilon_1)^2$.

Next, consider the nodes $u \in H$ which were moved to L in iteration j. By Lemma 5, $d_H(u) < l\cdot(1+\epsilon_1)$. So, the total number of edges attributed to u is less than $l\cdot(1+\epsilon_1)$.

Hence, the total number of edges attributed to all the nodes in H is less than $l\cdot(1+\epsilon_1)^2$. We replace $\deg(v)$ in the equation for the number of neighbors sampled by the number of attributed edges. Thus, the number of samples of attributed edges for any node in H is less than $\log_{1+\epsilon_1}(n/l) + \frac{2((1+c)\ln(n)+\ln(\log_{1+\epsilon_1} n))(1+\epsilon_1)^3}{\epsilon_1^3 l} l\cdot(1+\epsilon_1)^2$ $= \log_{1+\epsilon_1}(n/l) + \frac{2((1+c)\ln(n)+\ln(\log_{1+\epsilon_1} n))(1+\epsilon_1)^5}{\epsilon_1^3}$ which is dominated by $\log_{1+\epsilon_1}(n/l)+\frac{\ln(n)+\ln(\log_{1+\epsilon_1} n)}{\epsilon_1^3}$. We multiply this bound by n to get the bound when summed over all nodes and the total cost of sampling is bound by twice this, which is $2n\log_{1+\epsilon_1}(n/l) + \frac{2n(\ln(n)+\ln(\log_{1+\epsilon_1} n))}{\epsilon_1^3} = O(\frac{n\log n}{\epsilon^3})$.

Lines 6 to 17 of Algorithm 1 samples neighbors from nodes in H and performs a constant amount of work per sampled neighbor. In particular we note that in the adjacency list model, each execution of line 10 requires only constant time. Lines 18 to 25, for each node u in Q, perform a constant amount of work for every node that sampled u and each node in H can be added to the Q at most once. Thus, the running time of the randomized part of Algorithm 1 (Lines 3 to 31) is proportional to the number of samples.

Lemma 9. *Let $core(k)$ be the set of nodes in a k-core for some k. Let $outcore(k) = \{(u,v) \in E \mid u \in core(k), v \in \bigcup_{i\geq k} core(i)\}$. Then $|outcore(k)| \leq k\cdot|core(k)|$.*

Proof. In the peeling algorithm, after all nodes with core value $< k$ are peeled, every node remaining has degree at least k. When the nodes with core value k are peeled, every node peeled has degree no more than k when it is peeled. Therefore $|outcore(k)| \leq k\cdot|core(k)|$. ☐

The following lemma follows from Lemma 9.

Lemma 10. *The number of edges incident to nodes with core value no greater than k is no greater than $k\cdot|\bigcup_{k'\leq k} core(k')|$.*

When $p \geq 1$, $l = O(\log n)$. By the proof of Lemma 7, the core value of every node is $O(\log n)$ w.h.p. By Lemma 10, the number of edges incident to every node is $O(\log n)$. Since the peeling algorithm has running time linear in the number of edges, the running time of Line 32 of Algorithm 1 is $O(n\log n)$ w.h.p. Thus:

Lemma 11. *The running time of Algorithm 1 is $O(\frac{n\log n}{\epsilon^3})$.*

5.3 Running Time Comparison against Colton and Tsai Algorithm

The algorithm from [Farach-Colton and Tsai, 2016] uses an expected number of $\Theta(\frac{n\log n}{\epsilon^2})$ edges in the sampled subgraph, which is obtained in the streaming model by sampling each edge with probability $\Omega(\frac{n\log n}{\epsilon^2 m})$. Directly adapting this method to the adjacency list model would require scanning

[1]Proof details are available in the full version [King *et al.*, 2023]

over each edge. However, it can be modified to be more efficient for the adjacency list model by sampling a number of edges incident to each node according to a binomial distribution such that the resulting graph has the required number of edges. To compare the running time of our algorithm against the modified algorithm from [Farach-Colton and Tsai, 2016], we analyze the running time of Algorithm 1 with respect to the nodes that are sampled from, which are the nodes with degree $> \delta/(1+\epsilon_1)^2$, in Lemma 12.[1] We present a scenario where our algorithm performs asymptotically less sampling.

Lemma 12. *Given a graph $G = (V, E)$ with degeneracy δ, the running time of Algorithm 1 is $O(n + \frac{\lfloor D_s \rfloor}{\delta} \log n)$ where $D_s = \sum_{v \in N_s} deg(v)$ and N_s denotes the set of nodes in V with degree $> \delta/(1+\epsilon_1)^2$.*

The modified algorithm from [Farach-Colton and Tsai, 2016] has a running time of $O(n \log n)$ since it also runs proportionally to the number of sampled edges. Consider an input graph G, with n nodes, which consists of the union of disjoint cliques as follows: G has one larger clique of n^b nodes, where $b < 1$, and several other smaller cliques of $n^b/(1+\epsilon_1)^2$ nodes. We let B denote the set of nodes which belong to the larger clique. The degeneracy of G is $n^b - 1$ and Algorithm 1 only samples neighbors adjacent to the nodes in B which each have degree $n^b - 1$. So, we have that $D_s = n^b(n^b - 1)$. By Lemma 12, the running time of Algorithm 1 is $O(n^b \log n)$, which is asymptotically less than the running time of the algorithm from [Farach-Colton and Tsai, 2016].

6 Optimizations

In this section, we present optimizations to improve the running time of Algorithm 1 in practice.

6.1 Lower Starting Threshold

The initial degeneracy threshold of $l = \frac{n}{(1+\epsilon_1)}$ is too high in practice since it is unlikely that the degeneracy of the graph is n, and this will cause numerous iterations at the beginning of the algorithm where we remove all the nodes from H. We can improve this by lowering the starting threshold. Let d be the maximal integer where there are at least d nodes in G with degree d. The degeneracy of G is upper bounded by d. We can compute d once at the beginning of the algorithm, then divide l by $(1+\epsilon_1)$ and multiply p by $(1+\epsilon_1)$ until $l \leq d$.

6.2 Larger Threshold Leaps

We can further reduce the number of iterations required by changing l by more in each iteration where H becomes empty. That is, we divide l and multiply p by the sequence $(1+\epsilon_1), (1+\epsilon_1)^2, (1+\epsilon_1)^4, (1+\epsilon_1)^8, ...$ until we find a $(1+\epsilon_1)^{2^i}$ for which H is not empty at Line 26. Then, we binary search in the range $(1+\epsilon_1)^{2^{i-1}}$ to $(1+\epsilon_1)^{2^i}$ for the first k where the factor $(1+\epsilon_1)^k$ results in H not being empty.

6.3 Reusing Randomness

On Line 10, computing a random neighbor with replacement each time becomes very time consuming in practice. For the random sampling method to work, it does not require a new random number to be computed each time. In the correctness proof, in the proof of Lemma 4, since we take the union bound on the probability of error over all nodes and iterations, we can reuse the same randomness for sampling from all nodes in all iterations. At the beginning of the algorithm, we initialize an array R of size $md(V)$, where $md(V)$ is the maximum degree of any vertex in V of random numbers within the range $[0, 1)$. Then, each time we need to sample the i^{th} neighbor of a node v (Line 10), we take the $\lfloor R_i \cdot deg(v) \rfloor^{th}$ element from v's adjacency list. Since $k(v)$ increases in each iteration, we remember the sampled neighbors for v from the previous iteration and add to it as needed.

Graph	Nodes	Edges
clueweb12	978,408,098	74,744,358,622
uk-2014	787,801,471	85,258,779,845
gsh-2015	988,490,691	51,984,719,160
sk-2005	50,636,154	3,639,246,313
twitter-2010	41,652,230	2,405,026,390

Table 1: Summary of datasets

7 Computing Exact Degeneracy

We can extend Algorithm 1 to compute the exact degeneracy by running SemiDeg+. Since the output l of Algorithm 1 satisfies $\delta/(1+\epsilon) < l \leq \delta/(1+\epsilon)$ w.h.p., we can use $l \cdot (1+\epsilon)$ and $l/(1+\epsilon)$ as tighter upper and lower bounds in the SemiDeg+ algorithm binary search, which will will output the exact degeneracy with a smaller search range.

8 Experiments

We refer to our approach as NESD (Neighbor Sampling for Degeneracy).[2] We compare NESD (including all the optimizations from Section 6) to the approximation algorithm from [Farach-Colton and Tsai, 2016] and the state-of-the-art exact degeneracy algorithm, SemiDeg+, from [Li *et al.*, 2022]. Since the goal of our approach is to achieve an efficient sublinear $(1+\epsilon)$ approximation algorithm, we do not compare versus less accurate approximate algorithms nor other exact algorithms since they are superseded by SemiDeg+. We experiment with four different values of ϵ, namely 0.5, 0.25, 0.1, and 0.05, to illustrate the varying approximation accuracies of our algorithm. The value of c did not have a major impact on the results since the probability of correctness is high with n being large in all of the datasets. So, we use $c = 0.5$ for all experiments. We also show the running time of SemiDeg+ using the bounds from NESD (as described in Section 7), which we call NESD+, using the ϵ values of 0.1 and 0.05.

We use the datasets twitter-2010, sk-2005, gsh-2015, uk-2014, and clueweb12 from the Laboratory of Web Algorithmics [Boldi and Vigna, 2004; Boldi *et al.*, 2011; Boldi *et al.*, 2014], and the characteristic of each graph is illustrated in Table 1. We note that the number of edges shown in Table 1 is after symmetrization, where we add the reverse of directed

[2]An implementation of our method is available at https://github.com/QuintonYong/NESD

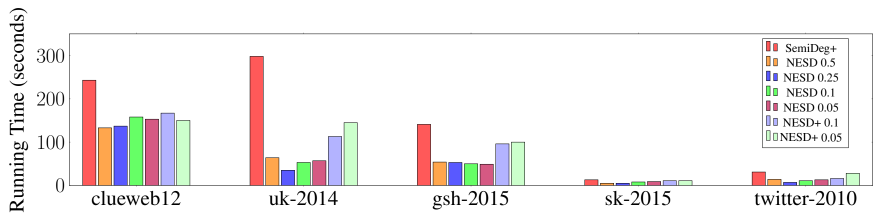

Figure 1: Running time comparison between SemiDeg+ and NESD with various ϵ and $c = 0.5$

	clueweb12		uk-2014		gsh-2015		sk-2005		twitter-2010	
	Output	Error	Output	Error	Output	Error	Output	Error	Output	Error
SemiDeg+	4244		10003		9888		4510		2488	
NESD 0.5	3696.835	0.871	8756.98	0.875	9418.101	0.952	4175.481	0.926	2162.936	0.869
NESD 0.25	4003.074	0.943	9884.616	0.988	9004.657	0.911	4338.047	0.962	2391.45	0.961
NESD 0.1	4186.726	0.987	9947.312	0.994	9601.476	0.971	4425.227	0.981	2376.771	0.955
NESD 0.05	4186.726	0.984	9850.165	0.985	9645.406	0.975	4452.054	0.987	2545.711	1.023

Table 2: Error factors of NESD degeneracy outputs with various ϵ

edges if they do not already exist, and without any self loops. Each of the graphs are massive webgraphs with over 1 billion edges, and the largest webgraph, clueweb12, has over 74 billion edges. The algorithms used in the experiments were implemented in Java 11, and the experiments were run on Amazon EC2 instances with the r5.24xlarge instance size (96 vCPU, 768GB memory) running Linux.

Figure 1 shows the running time results of NESD and NESD+. On each of the datasets, the approximation algorithm from [Farach-Colton and Tsai, 2016] was orders of magnitude slower than our algorithm and SemiDeg+ due to the requirement of passing through every single edge of the graph. As such, when run on massive webgraphs, the running time was extremely slow and we do not include the results in the figure. NESD ran 2.21x to 4.43x faster than SemiDeg+ on twitter-2010, 1.44x to 2.6x faster on sk-2005, 2.61x to 2.88x faster on gsh-2015, 4.66x to 8.51x faster on uk-2014, and 1.54x to 1.83x faster on clueweb12.

NESD+ with $\epsilon = 0.1$ was between 1.2x and 2.58x faster than SemiDeg+ on all datasets and NESD+ with $\epsilon = 0.05$ was between 1.11x and 2.06x faster than SemiDeg+ on all datasets. Furthermore, the total time of running NESD followed by NESD+ with $\epsilon = 0.1$ on the datasets uk-2014 and twitter-2010 was 1.8x and 1.15x faster than SemiDeg+ respectively, giving us a faster exact algorithm than the state-of-the-art on some datasets.

Table 2 shows the degeneracy δ of each dataset (output by SemiDeg+) as well as the approximate degeneracy and error factors from the output by NESD. The error factors on each dataset were between 0.869 and 0.952 for $\epsilon = 0.5$, between 0.911 and 0.988 for $\epsilon = 0.25$, between 0.955 and 0.994 for $\epsilon = 0.1$, and between 0.975 and 1.023 for $\epsilon = 0.05$. This shows that the error factors are well within the proven theoretical bounds and it also demonstrates the high level of degeneracy approximation accuracy of our algorithm.

9 Extension to k-Core Decomposition

We can extend Algorithm 1 to solve full k-core decomposition.[1] In this section, we describe how we can can modify the algorithm to label each node in the graph with an approximate core number within a factor of $(1 + \epsilon)$ of its true core number w.h.p. in $O(n \log n)$ time.

9.1 Algorithm Modifications

Instead of returning l as the approximate degeneracy when H is non-empty on Line 31, we continue the algorithm and assign an approximate core numbers to nodes which remain in H (Line 30). Within the body of the else statement on Line 30, we label all the nodes $v \in H$ with the value of l of that iteration. We then add the line of code $V \leftarrow L$ such that in the next iteration, the vertices in V are only the unlabelled nodes. We then also perform the same updates to l, p and $Sampled(v)$ for all $v \in V$. On Line 32, when $p \geq 1$, we run the peeling algorithm to on the remaining unlabelled nodes and assign core numbers to those nodes. Let l' be the last approximate label assigned within the while-loop. If the peeling algorithm assigns a core number of $> l'/(1 + (3/2)\epsilon_1)$ to a node, we instead label that node with l'. When all nodes have an assigned label, we return those labels as $(1 + \epsilon)$-approximate core numbers for each node.

10 Conclusion

We have devised $O(\frac{n \log n}{\epsilon^3})$ algorithms for degeneracy and for k-core in the adjacency list model. Both algorithms give provably close approximate solutions, w.h.p. We have shown in experiments, that on all our instances of very large graphs, our algorithm for degeneracy, NESD works within our approximation guarantees, and for all our datasets, it gives a significant speed up over prior state-of-the-art methods for computing degeneracy. Our algorithm features various optimizations which may be of independent interest.

Acknowledgements

Valerie King, Alex Thomo, and Quinton Yong thank NSERC for their support.

References

[Aamand *et al.*, 2021] Anders Aamand, Adam Karczmarz, Jakub Lacki, Nikos Parotsidis, Peter M. R. Rasmussen, and Mikkel Thorup. Optimal decremental connectivity in non-sparse graphs. *CoRR*, abs/2111.09376, 2021.

[Altaf-Ul-Amin *et al.*, 2006] Md Altaf-Ul-Amin, Yoko Shinbo, Kenji Mihara, Ken Kurokawa, and Shigehiko Kanaya. Development and implementation of an algorithm for detection of protein complexes in large interaction networks. *BMC bioinformatics*, 7(1):207–207, 2006.

[Alvarez-Hamelin *et al.*, 2005] José Alvarez-Hamelin, Luca Dall'Asta, Alain Barrat, and Alessandro Vespignani. K-core decomposition: A tool for the visualization of large scale networks. *Adv. Neural Inf. Process. Syst.*, 18, 04 2005.

[Bahmani *et al.*, 2012] Bahman Bahmani, Ravi Kumar, and Sergei Vassilvitskii. Densest subgraph in streaming and mapreduce. *Proc. VLDB Endow.*, 5(5):454–465, jan 2012.

[Batagelj and Zaversnik, 2003] Vladimir Batagelj and Matjaz Zaversnik. An o(m) algorithm for cores decomposition of networks. *CoRR*, cs.DS/0310049, 2003.

[Bhattacharya *et al.*, 2015] Sayan Bhattacharya, Monika Henzinger, Danupon Nanongkai, and Charalampos Tsourakakis. Space- and time-efficient algorithm for maintaining dense subgraphs on one-pass dynamic streams. In *Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '15, page 173–182, New York, NY, USA, 2015. Association for Computing Machinery.

[Bhawalkar *et al.*, 2012] Kshipra Bhawalkar, Jon Kleinberg, Kevin Lewi, Tim Roughgarden, and Aneesh Sharma. Preventing unraveling in social networks: The anchored k-core problem. In Artur Czumaj, Kurt Mehlhorn, Andrew Pitts, and Roger Wattenhofer, editors, *Automata, Languages, and Programming*, pages 440–451, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[Boldi and Vigna, 2004] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[Boldi *et al.*, 2011] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar, editors, *Proceedings of the 20th international conference on World Wide Web*, pages 587–596. ACM Press, 2011.

[Boldi *et al.*, 2014] Paolo Boldi, Andrea Marino, Massimo Santini, and Sebastiano Vigna. BUbiNG: Massive crawling for the masses. In *Proceedings of the Companion Publication of the 23rd International Conference on World Wide Web*, pages 227–228. International World Wide Web Conferences Steering Committee, 2014.

[Brandeis *et al.*, 2020] Simon Brandeis, Adrian Jarret, and Pierre Sevestre. About graph degeneracy, representation learning and scalability. *ArXiv*, abs/2009.02085, 2020.

[Chazelle *et al.*, 2005] Bernard Chazelle, Ronitt Rubinfeld, and Luca Trevisan. Approximating the minimum spanning tree weight in sublinear time. *SIAM journal on computing*, 34(6):1370–1379, 2005.

[Cheng *et al.*, 2011] James Cheng, Yiping Ke, Shumo Chu, and M. Tamer Özsu. Efficient core decomposition in massive networks. In *2011 IEEE 27th International Conference on Data Engineering*, pages 51–62, 2011.

[Dvorak, 2013] Zdenek Dvorak. Constant-factor approximation of the domination number in sparse graphs. *European journal of combinatorics*, 34(5):833–840, 2013.

[Eden *et al.*, 2022] Talya Eden, Saleet Mossel, and Dana Ron. Approximating the arboricity in sublinear time. In *SODA*, 2022.

[Eppstein and Strash, 2011] David Eppstein and Darren Strash. Listing all maximal cliques in large sparse real-world graphs. In *EXPERIMENTAL ALGORITHMS*, volume 6630 of *Lecture Notes in Computer Science*, pages 364–375, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[Esfandiari *et al.*, 2018] Hossein Esfandiari, Silvio Lattanzi, and Vahab Mirrokni. Parallel and streaming algorithms for k-core decomposition. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1397–1406. PMLR, 10–15 Jul 2018.

[Farach-Colton and Tsai, 2014] Martín Farach-Colton and Meng-Tsung Tsai. Computing the degeneracy of large graphs. In *LATIN 2014: Theoretical Informatics*, Lecture Notes in Computer Science, pages 250–260. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[Farach-Colton and Tsai, 2016] Martín Farach-Colton and Meng-Tsung Tsai. Tight approximations of degeneracy in large graphs. In Evangelos Kranakis, Gonzalo Navarro, and Edgar Chávez, editors, *LATIN 2016: Theoretical Informatics*, pages 429–440, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[Ghaffari *et al.*, 2019] Mohsen Ghaffari, Silvio Lattanzi, and Slobodan Mitrović. Improved parallel algorithms for density-based network clustering. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 2201–2210. PMLR, 09–15 Jun 2019.

[Goodrich and Pszona, 2011] Michael T. Goodrich and Paweł Pszona. External-memory network analysis algorithms for naturally sparse graphs. In *Algorithms – ESA 2011*, Lecture Notes in Computer Science, pages 664–676. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[King *et al.*, 2023] Valerie King, Alex Thomo, and Quinton Yong. Computing (1+epsilon)-approximate degeneracy in sublinear time. *ArXiv*, abs/2211.04627, 2023.

[Lenzen and Wattenhofer, 2010] Christoph Lenzen and Roger Wattenhofer. Minimum dominating set approximation in graphs of bounded arboricity. In *DISTRIBUTED COMPUTING*, volume 6343 of *Lecture Notes in Computer Science*, pages 510–524, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[Li *et al.*, 2014] Rong-Hua Li, Jeffrey Xu Yu, and Rui Mao. Efficient core maintenance in large dynamic graphs. *IEEE transactions on knowledge and data engineering*, 26(10):2453–2465, 2014.

[Li *et al.*, 2022] Rong-Hua Li, Qiushuo Song, Xiaokui Xiao, Lu Qin, Guoren Wang, Jeffrey Xu Yu, and Rui Mao. I/o-efficient algorithms for degeneracy computation on massive networks. *IEEE Transactions on Knowledge and Data Engineering*, 34(7):3335–3348, 2022.

[Lick and White, 1970] Don R. Lick and Arthur T. White. k-degenerate graphs. *Canadian Journal of Mathematics*, 22(5):1082–1096, 1970.

[Limnios *et al.*, 2021] Stratis Limnios, George Dasoulas, Dimitrios M. Thilikos, and Michalis Vazirgiannis. Hcore-init: Neural network initialization based on graph degeneracy. In *2020 25th International Conference on Pattern Recognition (ICPR)*, pages 5852–5858, 2021.

[Matula and Beck, 1983] David Matula and Leland Beck. Smallest-last ordering and clustering and graph coloring algorithms. *Journal of the ACM*, 30(3):417–427, 1983.

[Nikolentzos *et al.*, 2018] Giannis Nikolentzos, Polykarpos Meladianos, Stratis Limnios, and Michalis Vazirgiannis. A degeneracy framework for graph similarity. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 2595–2601. International Joint Conferences on Artificial Intelligence Organization, 7 2018.

[Saríyüce *et al.*, 2013] Ahmet Erdem Saríyüce, Buğra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Umit V. Çatalyurek. Streaming algorithms for k-core decomposition. *Proceedings of the VLDB Endowment*, 6(6):433–444, 2013.

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

[Wen *et al.*, 2015] Dong Wen, Lu Qin, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. I/O efficient core graph decomposition at web scale. *CoRR*, abs/1511.00367, 2015.