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

by

Quinton Yong

B.Sc. (Computer Science), University of Victoria, 2020

A Thesis Submitted in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

© Quinton Yong, 2022

University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.

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

by

Quinton Yong

B.Sc. (Computer Science), University of Victoria, 2020

Supervisory Committee

Dr. V. King, Supervisor

(Department of Computer Science)

Dr. A. Thomo, Supervisor

(Department of Computer Science)

Supervisory Committee

Dr. V. King, Supervisor

(Department of Computer Science)

Dr. A. Thomo, Supervisor

(Department of Computer Science)

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 on a graph with n nodes, sublinear in the input size for dense graphs, by sampling a small number of neighbours adjacent to high degree nodes. Our algorithm can also be extended to an $O(n \log n)$ time solution to the k-core decomposition problem. This improves upon the method by Bhattacharya et al., which implies a $(4 + \epsilon)$-approximate $\tilde{O}(n)$ solution to the degeneracy problem. Our techniques are similar to other sketching methods which use sublinear space for k-core and degeneracy. 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, including the 2022 exact degeneracy algorithm by Li et al.

Table of Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I would like to thank:

My family, for their constant love and support along my academic journey

Dr. King, for her exceptional mentorship and sharing her knowledge in theoretical computer science

Dr. Thomo, for his incredible guidance, encouragement, and sharing his expertise in the topic of my degree

Chapter 1

Introduction

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 induced subgraph in which all nodes have degree at least k, and the 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 [21, 12], approximating the maximum density of any subgraph within a constant factor [10, 17], approximating the arboricity of a graph within a constant factor [9], and maximal clique enumeration [15].

The k-core decomposition problem, and consequently the degeneracy problem, can be solved using a well known linear time "peeling" algorithm [25]. Starting with $k = 1$, the algorithm repeatedly removes all nodes in G 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 [3], with fully

external-memory [11], and with semi external-memory [28]. There are also algorithms for approximate computing of k-core in dynamic and streaming settings [23, 26, 16]. In this work, we focus on degeneracy computation where the graph is given as an adjacency list in memory.

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, including the SemiDeg+ algorithm from [22], which is the state-of-the-art in exact degeneracy computation. In a streaming / semi-streaming model, [17], [20], and [2] propose $(2+\epsilon)$-approximate solutions and [18] proposes a $(1+\epsilon)$-approximate solution. However, in an adjacency list model, these solutions require each edge to be scanned. A sublinear time algorithm for degeneracy in this model is the algorithm presented in [5] for computing the maximum density of any subgraph. It runs in $\tilde{O}(n)$ time and implies a $(4+\epsilon)$-approximate algorithm for degeneracy.

In this work, we present a $(1+\epsilon)$-approximate $O(n \log n)$ 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). Our algorithm can be extended to a sublinear time algorithm for k-core decomposition, but the time savings over other algorithms are particularly exhibited empirically for the degeneracy problem. We avoid the overhead of full k-core decomposition and directly compute degeneracy of the given graph G by sampling the neighbours 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, then we output l as the (approximate) degeneracy of G. We prove the theoretical guarantees of our algorithm and we also propose opti-

mizations which improve its practical running time. We also compare our solution to the $(1+\epsilon)$-approximate algorithm of [18] and the SemiDeg+ algorithm from [22]. We show that, on massive real-world webgraphs, our algorithm is substantially faster than both on almost all datasets. We also compare our algorithm to the algorithm from [18] modified for the adjacency list model and show that our algorithm can run asymptotically faster.

The main result of this work is the following theorem:

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

Chapter 2

Background

This chapter presents the necessary background, definitions, and notation used in this thesis, as well as the statement of the problem solved by this work. Section 2.1 presents the definitions and notations, Section 2.2 discusses the input models used by this work and related work, and Section 2.3 presents the problem definition.

2.1 Definitions and Notation

Graph. A *graph*, denoted $G = (V, E)$ is a structure where a set of objects are related to one another. The objects are called **vertices** or **nodes**, denoted by V, and the set of relations are called **edges**, denoted by E. The number of vertices is denoted by n and the number of edges is denoted by m.

Neighbour. A vertex u is a *neighbour* of a vertex v if u and v are connected by an edge. The set of neighbours of a vertex v is denoted $N(v)$.

Degree. The *degree* of a vertex v is the number of v's neighbours, denoted $deg(v)$.

Undirected Graph. Graphs for which edges do not have a direction are called *undirected graphs*. In this work, the graphs are undirected unless specified otherwise.

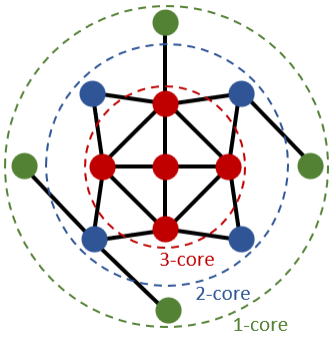

Figure 2.1: The k-core decomposition of a graph

Undirected Edge. An edge in an undirected graph which connects a vertex u and a vertex v is called an *undirected edge*, and is denoted by (u, v). Note that (u, v) in this case denotes an unordered pair.

Subgraph. A graph $G' = (V', E')$ is a *subgraph* of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. The degree of a vertex v in G' is denoted as $d_{G'}(v)$.

Induced Subgraph. A subgraph induced by a subset of nodes $V' \subseteq V$ is denoted $G_{V'} = (V', E')$ where E' is defined as all edges $(u, v) \in E$, where both u and v are in V'.

Clique. A *clique* in a graph is a subset of vertices such that every two distinct vertices are adjacent to one another.

k-core. For a graph G and an integer k, a *k-core* of G is a maximal induced subgraph of G such that every node in the subgraph has degree at least k.

k-core decomposition. The *k-core decomposition* of a graph G is an enumeration of all the k-cores of G (illustrated in Figure 2.1).

Core Number. The *core number* or core value of a vertex $v \in V$, denoted $c(v)$, is the largest k for which there is a k-core that contains v.

Degeneracy. The *degeneracy* of a graph G, denoted δ, is equal to the largest k for which G contains a non-empty k-core (i.e. the degeneracy equals the maximum core number). The degeneracy of G is also defined as the smallest integer such that every non-empty induced subgraph has a vertex with degree at most δ.

Approximate Algorithm. An γ-approximate algorithm outputs a solution l which satisfies $\delta/\gamma \leq l \leq \delta \cdot \gamma$, where δ is the true solution and γ is called the error factor.

Symbols	**Definitions**
G	An undirected graph.
V	The set of vertices.
E	The set of edges.
n	The number of vertices. $n = \|V\|$.
m	The number of edges. $m = \|E\|$.
u, v	Reserved for vertices.
$N(v)$	The neighbours of vertex v.
$deg(v)$	The degree of vertex v.
$d_{G'}(v)$	The degree of a vertex v in a subgraph G'.
(u, v)	An undirected edge between u and v.
$c(v)$	The core number of a vertex v
δ	The degeneracy of the graph.
i, j	Reserved for iteration numbers.
ϵ	An error parameter given as input.
ϵ_1	Assigned the value $\epsilon/3$ in our algorithm.
c	A constant given as input.
p	Reserved for probabilities.
l	The output of our algorithm.
H	Set of high degree nodes.
L	Set of low degree nodes.
Q	A queue of nodes.

Table 2.1: Commonly used notation in this thesis

2.2 Graph Input Model

The graph input model used by this work is called the *adjacency list model*, where each node $v \in V$ has an adjacency list of its neighbours in an arbitrary order. In this model, there are two types of queries with $O(1)$ complexity that are used to access graph information:

1. **Degree queries:** get the degree of any node $v \in V$
2. **Neighbour queries:** get the i^{th} neighbour of any node $v \in V$, which is the i^{th} element in v's adjacency list

The adjacency list model is a standard graph representation which is often used in sublinear time algorithms.

A graph model used by related work is the *streaming model*, where the vertices in the graph are known immediately, but the edges arrive one at a time in a stream. Algorithms which use the streaming model are only allowed a small amount of memory (usually logarithmic in n and m) and aim to process each edge quickly and make a small constant number of passes over the input stream. The *semi-streaming* model is a similar model where the amount of memory allowed is instead $O(n \text{ polylog } n)$. *Sketching* is a term used with streaming and semi-streaming graph models, where the algorithm maintains a small portion (sketch) of the graph following the memory constraints and uses the sketch to approximate a solution to the problem. Another graph model is the *dynamic* model where edges can be either inserted or deleted over time.

The large graphs used in this work are stored using the WebGraph framework [8], which uses several compression techniques to compress the adjacency list representation of large web graphs (graphs which represent web pages and links to other web pages). The main idea is that instead of storing the ID's of the neighbour nodes

(where nodes are numbered from 0 to $n-1$), the gaps between the ID numbers are stored. This works well with web graphs since the nodes are numbered according to the lexicographic ordering of URLs, so the gaps result in smaller numbers. The WebGraph framework further compresses the graph using techniques such as representing a node v's edges as a sequence of bits which represent the shared edges between v and another reference node r, then storing the number of consecutive 1's and 0's in the bit sequence instead of the sequence itself.

Although compressed graphs have a significantly reduced storage size, performing neighbour queries requires decompression. Thus, as a preprocessing step in our implementation, we convert the WebGraph graph into an array representation of the adjacency list model (which we do not consider as part of the running time of our algorithm), similar to the graph representation described in Section 3.1.

2.3 Problem Definition

The goal of the work presented in this thesis is to design an efficient algorithm for approximating the degeneracy δ of a given graph $G = (V, E)$ in an uncompressed adjacency list form within a factor of $(1+\epsilon)$ with high probability, where ϵ is an input parameter in the range $(0, 1]$. We define high probability as occurring with probability at least $1 - \frac{2}{n^c}$, where $n = |V|$ and c is a constant given as an input parameter.

Chapter 3

Related Work

This chapter presents the work in degeneracy computation which is most relevant to our algorithm. The first algorithm discussed is the simplest algorithm for k-core decomposition, which gives a linear time algorithm for degeneracy, followed by the current state-of-the-art algorithm for exact degeneracy computation. Then, we discuss approximate algorithms for degeneracy computation, which do not use the adjacency list model for graph input, but contain techniques which are important in our work. Lastly, we discuss algorithms which use different techniques than our work but are closely related to the degeneracy problem.

3.1 Peeling Algorithm

The simplest method to compute the k-core of a graph is to repeatedly remove all nodes (and their incident edges) with degree less than k until all remaining nodes have degree at least k [27]. This idea can be used to enumerate all k-cores of the graph by starting with $k = 1$ (or the the degree of the minimum degree node), removing all nodes with degree less than k, then incrementing k and repeating the process until a k is reached where all nodes are removed from the graph. This is known as the

peeling algorithm and was originally introduced by Matula and Beck in 1983 [25]. The pseudocode is illustrated in Algorithm 1. The peeling algorithm can also be modified to verify if a k-core exists for a particular k by repeatedly removing all nodes with degree less than k then verifying if there are nodes that remain.

It was shown by Batagelj and Zaversnik in 2003 [4] that this algorithm can be implemented in linear time by maintaining bins of nodes for each possible degree in the graph (from 0 to the maximum degree). This is done by storing all nodes in an array sorted by bins and the boundary between bins are maintained in a separate array. The edges are represented using the adjacency list model implemented as a flat array of size $2m$ where neighbours of a node are sequentially stored and the boundaries between groups of neighbours are maintained in a separate array. This way, when removing removing a node v, removing an edge (u, v) and decrementing the degree of v can be done by a swap operation and updating the bin boundary. Since the maximum possible degree is $n-1$, initially ordering the vertices takes $O(n)$ time, and removing an edge takes $O(1)$ time. Therefore, the algorithm runs in $O(m+n)$ time. The peeling algorithm is thus a straightforward linear time algorithm for computing the degeneracy of a graph by computing all k-cores of a graph then returning the largest k for which the graph has a k-core.

Although the peeling algorithm is fairly efficient, especially for smaller graphs, requiring linear time in the number of nodes and edges makes it slow for large graphs and much work has been done to improve improve the practical running time of k-core enumeration. More importantly, since we are only interested in the degeneracy of a graph, full k-core algorithms induce the unnecessary overhead of computing all k-cores instead of just finding the maximum k-core.

Algorithm 1 Peeling Algorithm

Input: Graph $G = (V, E)$

Output: Core numbers of each vertex $v \in V$

```
1: Order the vertices v ∈ V in increasing order of their degrees
2: for v ∈ V in order do
3:     c(v) ← deg(v)
4:     for all u ∈ N(v) do
5:         if deg(u) > deg(v) then
6:             deg(u) ← deg(u) − 1
7:             Reorder V accordingly
```

3.2 SemiDeg Algorithm

The SemiDeg algorithm [22] utilizes binary search within a maintained upper and lower bound of the degeneracy to achieve an efficient practical degeneracy algorithm. It uses h-index as a primary tool to compute upper bounds, where the h-index of a set of numbers is defined as the largest k such that there are k values in the set greater than or equal to k and the h-index of a node is defined as the h-index of the set of degrees of its neighbours. An important observation is that the core number of a node is equal to the h-index of the core numbers of the node's neighbours. Also, the h-index of a set of upper bounds on core numbers of a node v's neighbours (for example their degrees) is an upper bound of v's core number. It follows that the h-index of the set of degrees of nodes in V is an upper bound of the degeneracy, which is denoted $h(\mathbf{d})$. A tighter upper bound of the degeneracy, denoted h^*, is the h-index of the set of h-indices of all nodes in V, in other words the largest k such that there are k nodes each with k neighbours of degree k. From [24], it is known that $\lceil \frac{m}{n-1} \rceil$ is a lower bound for the degeneracy. SemiDeg performs a binary search within the

range $[\lceil \frac{m}{n-1} \rceil, h^*]$ to find the exact degeneracy by finding the largest k for which the graph contains a k-core using the peeling algorithm for $k \in [\lceil \frac{m}{n-1} \rceil, h^*]$.

However, computing h^* and using the peeling algorithm to test each k is inefficient. The paper [22] also proposes an improved algorithm called SemiDeg+. Initially, the upper bound used is $h(\mathbf{d})$ rather than h^*. Then instead of performing the peeling algorithm on the whole graph to check if particular k-core exists, it uses the observation that if a k-core exists, it must be in the subgraph induced by the nodes whose upper bound on core value is at least k. So, when testing for a k-core, SemiDeg+ uses the subroutine called PCore which only looks at the induced subgraph as described, and at the same time updates the upper bound of the core values of those nodes. Initially, the upper bound of core value of all nodes is their degree, and they are updated in the PCore subroutine to be the h-index of the set of upper bounds on core values of their neighbours.

Although SemiDeg+ has state-of-the-art practical running time, the running time depends on each input graph. Specifically, the running time of SemiDeg+ is $O(h(\mathbf{d}) \times (\tilde{n} + \tilde{m}) \log n)$, where $h(\mathbf{d})$ is as defined above, $\tilde{n}$ is the maximum number of of nodes given to the PCore subroutine, and $\tilde{m}$ is the total number of edges incident to the nodes given to the PCore subroutine. We show in Chapter 8 that, in practice, our algorithm runs faster than SemiDeg+ on large graphs.

3.3 Colton and Tsai Algorithm

Colton and Tsai [18] present a randomized algorithm which approximates the degeneracy of a graph within an error factor of $(1 + \epsilon)$. This improves upon their previous work in [17], which approximated the degeneracy within an error factor of $(2 + \epsilon)$. The paper [18] shows that computing a subgraph resulting from sampling each edge with probability $\Omega(\frac{n \log n}{\epsilon^2 m})$ then computing the degeneracy of that subgraph

using the peeling algorithm results in a $(1+\epsilon)$-approximation of the true degeneracy with probability $1 - \frac{1}{n^{\Theta(1)}}$. In other words, the expected number of edges required is $\Theta(\frac{n \log n}{\epsilon^2})$.

The algorithm is presented in a semi-streaming model, where the nodes in the graph are known ahead of time, but the edges are given in a stream. So, n is known ahead of time, but m is unknown until the end of the stream. However, since we know that the number of sampled edges required is $\Theta(\frac{n \log n}{\epsilon^2})$, the algorithm starts by guessing $m = \Theta(\frac{n \log n}{\epsilon^2})$ and samples each edge with probability $p = 1$. Then, if it is discovered that there are more than m edges, m is doubled and p is halved. Thus m and p are continuously adjusted, and edges are resampled / discarded, until the end of the stream. In the adjacency list model, there is no need for the procedure to guess m since the graph is known and we can simply sample each edge with probability $\Omega(\frac{n \log n}{\epsilon^2 m})$.

The correctness of the algorithm comes from being able to bound the degree of nodes when the number of edges in the graph is around $O(n \log n)$, which is similar to our method. However, there is a significant overhead required to look at and randomly sample every edge, compared to our method which only looks at a small number of neighbours of nodes (i.e. edges). An adaptation of this algorithm to the adjacency list model, which is not included in [18], would be to sample the edges incident to each node with a probability distribution such that the resulting graph has the expected $\Theta(\frac{n \log n}{\epsilon^2})$ edges as needed. This would reduce the overhead required to look at every edge, but we show in Section 5.2.1 that our method, which gradually finds the sampling probability, can result in asymptotically less sampling.

3.4 Sketching Algorithm

Esfandiari et al. [16] present a $(1+\epsilon)$-approximate sketching algorithm for k-core decomposition where graph input is given as a stream of edges, similar to the Colton and Tsai Algorithm, but first we describe the algorithm as if the graph was given in the adjacency list model. The algorithm performs $\log n$ rounds of sampling edges of G with probability p, starting with probability $p = O(\frac{\log n}{\epsilon^2 n})$ and doubling at the end of each round. In the algorithm, p is intialized as $\frac{96 \log n}{\epsilon^2 n}$. At the end of an iteration, nodes with sufficiently high core numbers in the subgraph induced by sampled edges are assigned an approximate core number label. Then, the edges induced by the set of labeled nodes are removed from the sampling. Every round thus produces sparse induced subgraphs to keep memory usage low, and the idea is that if the core number of a node in a particular subgraph is high enough despite the low sampling probability, then it is possible to estimate its core number. The peeling algorithm is used to obtain the core number of each node v (which has not been assigned an approximate core number) within each induced subgraph, which is denoted $l_j(v)$ where j is the round number. If $l_j(v)$ for an a node v is sufficiently high, it is assigned an approximate core number. Specifically, if $l_j(v)$ is between $\frac{192 \log n}{\epsilon^2}$ and $\frac{384 \log n}{\epsilon^2}$, v is assigned an approximate core number of $(1-\epsilon)\frac{l_j(v)}{p_j}$, where p_j is the sampling probability in round j. If $l_j(v) > \frac{384 \log n}{\epsilon^2}$, then v is assigned an approximate core number of $\frac{2(1-\epsilon)n}{2^{j-1}}$.

When the graph input is a stream of edges, the algorithm maintains the $\log n$ sampled subgraphs for each of the $\log n$ rounds. Whenever an edge is inserted, it is sampled with each probability for each iteration until it is removed (in the case where both of its endpoints have been assigned an approximate core number). In rounds where the new edge is sampled and added, the core numbers of nodes in the associated subgraphs are updated.

The algorithm works without using too much memory since each subgraph is

small and it is shown in [16] that the total space used by the algorithm is $O(\frac{n \log^2 n}{\epsilon^2})$. Also, in a parallel computing environment, the step of sampling edges can be done in parallel over multiple machines. However, in the adjacency list model, the algorithm suffers in terms of running time due to the requirement of scanning and sampling each edge (which induces an $O(m)$ complexity overhead), as well as running the peeling algorithm for each subgraph. Our work uses a similar idea of using increasing sampling probabilities to determine the core number of nodes with high probability, however we significantly reduce the running time by only looking at a small number of edges when sampling neighbours.

3.5 Other Related Algorithms

Ghaffari et al. [19] improves the work of [16] in a parallel computing environment. The algorithm reduces the number of parallel computation rounds required, while maintaining the $(1 + \epsilon)$ approximation ratio, and uses a vertex sampling technique instead of sampling edges.

Bhattacharya et al. [5] present a $(2 + \epsilon)$-approximate sublinear time algorithm for the densest subgraph problem, where the density of a graph is defined as the number of edges divided by the number of vertices, and the problem is to find the subgraph of a given graph which produces the maximum density, whose density is denoted d^*. This work implies a $(4 + \epsilon)$-approximate sublinear time algorithm for degeneracy computation since the degeneracy is a 2-approximation for the density of the densest subgraph [10]. The algorithm in [5] samples edges to obtain a sketch of the input graph in order to approximate an extension of a k-core called a (α, k, L)-decomposition, where $\alpha > 1$ and the algorithm uses $L = O(\log_{(1+\epsilon)} n)$. The (α, k, L)-decomposition of a graph $G = (V, E)$ is an approximate version of the k-core where we run L iterations of removing all nodes with degree less than k and some nodes with

degree between k and αk, and the decomposition is the tuple $(Z_1, ..., Z_L)$ where each Z_i are the nodes which remain in each iteration $i \in [1, L]$ and $Z_1 = V$. A theorem of [5] shows that the largest k which results in a non-empty Z_L in the (α, k, L)-decomposition is a $(2+\epsilon)$ approximation for d^*. The algorithm finds the largest such k by trying different k in powers of $(1+\epsilon)$.

Another related problem is finding the arboricity of a graph, where arboricity is defined as the minimum number of forests for which the graph can be partitioned into. The aboricity, denoted arb, and the degeneracy δ of a graph are closely related since it is known that $arb \leq \delta \leq 2 \cdot arb$ [14]. Eden et al. [13] present a $O(\log^2 n)$-approximate sublinear time algorithm in the adjacency list model which uses a technique similar to the decomposition method from [5].

Chapter 4

Degeneracy Algorithm

In this section, we present the main algorithm of this work starting with a high level description followed by the detailed description. We re-state that the goal of the algorithm is to compute a $(1+\epsilon)$-approximation of the degeneracy of a graph with high probability and with a fast running time.

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 approximate the degeneracy of the graph by looking at the subgraph induced by high degree nodes, which we denote as H.

Let l be the value which we approximate to be the degeneracy of the graph. Initially, all nodes in V are put into H if their degree is above the threshold l, and the algorithm performs iterations where we check if H contains an l-core (within an error factor of $(1+\epsilon)$) with high probability. In each iteration, we decrease l when there is a high probability that H does not contain an l-core.

To efficiently check if H does contain an l-core, we sample a small number of

neighbours incident 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 neighbours that are also in H. If it does not sample a sufficient number of neighbours in H, then it is likely not part of an l-core, and we remove the node from H and check its neighbours and remove them from H if necessary (similar to the peeling algorithm). If no nodes remain unpeeled, then we lower the threshold l and repeat the procedure. Otherwise, we conclude that H is an approximate l-core, and since this is the largest core found thus far, we output l as the approximate degeneracy of G. In Chapter 5, we prove that the output is within a factor of $(1+\epsilon)$ of the true degeneracy of G with probability at least $1 - \frac{2}{n^c}$.

4.2 Algorithm Details

The approximate algorithm for computing degeneracy is illustrated in Algorithm 2. In each iteration of the while-loop at Line 5 of Algorithm 2, 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 $L = V \setminus H$ (Algorithm 2 Lines 6 and 7). Throughout the algorithm, we use ϵ_1 which is set to $\epsilon/3$, which is explained further in Chapter 5. Initially, $l = \frac{n}{(1+\epsilon_1)}$, and we decrease l by a factor of $(1+\epsilon_1)$ at the end of each iteration when we do not return an approximate degeneracy (Algorithm 2 Lines 2 and 12).

We then sample a small number of neighbours for each node in H (Algorithm 3 Line 5). Specifically, we sample $k(v)$ neighbours u of v with replacement from every node $v \in H$, where $k(v) = \lceil p \cdot \deg(v) \rceil$. 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 (Algorithm 2 Lines 3 and 13).

Next, we check if H contains an l-core with high probability. 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 neighbours in H. In particular, each node $v \in H$ is assigned a value $t(v)$ which is initially set to the number of neighbours sampled by v, $k(v)$, and represents the degree of v in H (Algorithm 3 Line 3). Since we sampled $k(v)$ out of $deg(v)$ neighbours, if H is an l-core, then we expect that each node v samples at least $l \cdot k(v)/deg(v)$ neighbours which are also in H. So, if at any point $t(v)$ of a node $v \in V$ is less than $l \cdot k(v)/deg(v)$, then v is peeled from H to L (Algorithm 3 Line 8 and Algorithm 4 Line 6).

During the sampling process (Algorithm 3 Lines 4 - 13), whenever a node $v \in H$ samples a neighbour $u \in L$, we decrement $t(v)$ (Algorithm 3 Line 7). Otherwise, if it samples a neighbour $u \in H$, we need to remember that u was sampled by v so if u is peeled from H later, we need to decrement $t(v)$. Each node $u \in H$ maintains an initially empty list (with possible repeats), denoted $Sampled(u)$, containing the nodes which sampled them. If a node $v \in H$ samples a neighbour $u \in H$, we add v to $Sampled(u)$ (Algorithm 3 Line 13). Then, if a node $u \in H$ is peeled due to $t(u)$ falling below $l \cdot k(u)/deg(u)$, we add u to an initially empty queue Q (Algorithm 3 Line 11).

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 (Algorithm 4 Line 5). Similarly, if $t(v)$ for a node $v \in H$ is less than $l \cdot k(v)/deg(v)$, then v is moved to L and is also added to Q (Algorithm 4 Lines 7 - 9).

If there are no remaining nodes in H, that means each node that was initially 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 $l \leftarrow l/(1+\epsilon_1)$ and $p \leftarrow p \cdot (1+\epsilon_1)$, and we also reset the sampled lists to empty (Algorithm 2 Lines 12 - 14). Otherwise, if there are nodes remaining in H,

then we expect that those nodes form 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 (Algorithm 2 Line 16). In the case where $p \geq 1$ before l is returned, we simply compute δ by running the peeling algorithm on G and return δ (Algorithm 2 Lines 17 - 18).

Algorithm 2 ApproximateDegeneracy(G, ϵ, c)

Input: Graph $G = (V, E)$, error factor $\epsilon \in (0, 1]$, constant c

Output: $(1 + \epsilon)$-approximate degeneracy of G with probability $\geq 1 - \frac{2}{n^c}$

1: $\epsilon_1 \leftarrow \epsilon/3$
2: $l \leftarrow \frac{n}{(1+\epsilon_1)}$
3: $p \leftarrow \frac{2((1+c)\ln(n)+\ln(\log_{1+\epsilon_1}(n)))(1+\epsilon_1)^2}{\epsilon_1^2 n}$
4: $Sampled(u) \leftarrow \emptyset$ for all $u \in V$
5: **while** $p < 1$ **do**
6: $\quad H \leftarrow \{v \in V \mid deg(v) \geq l\}$
7: $\quad L \leftarrow V \setminus H$
8: $\quad Q \leftarrow \emptyset$
9: $\quad$ SampleHighDegreeNodes()
10: $\quad$ PeelLowDegreeNodes()
11: $\quad$ **if** $H = \emptyset$ **then**
12: $\quad\quad l \leftarrow l/(1 + \epsilon_1)$
13: $\quad\quad p \leftarrow p \cdot (1 + \epsilon_1)$
14: $\quad\quad Sampled(u) \leftarrow \emptyset$ for all $u \in V$
15: $\quad$ **else**
16: $\quad\quad$ **return** l
17: compute δ by running the peeling algorithm
18: **return** δ

Algorithm 3 SampleHighDegreeNodes()

for each node $v \in H$ **do**
  $k(v) \leftarrow \lceil p \cdot deg(v) \rceil$
  $t(v) \leftarrow k(v)$
  for $i \in 1, ..., k(v)$ **do**
    pick a neighbour u of v independently at random
    if $u \in L$ **then**
      $t(v) \leftarrow t(v) - 1$
      if $t(v) < l \cdot k(v)/deg(v)$ **then**
        $H \leftarrow H \setminus \{v\}$
        $L \leftarrow L \cup \{v\}$
        add v to Q
    else
      add v to $Sampled(u)$

Algorithm 4 PeelLowDegreeNodes()

while $Q \neq \emptyset$ **do**
  $u \leftarrow dequeue(Q)$
  for each node $v \in Sampled(u)$ **do**
    if $v \in H$ **then**
      $t(v) \leftarrow t(v) - 1$
      if $t(v) < l \cdot k(v)/deg(v)$ **then**
        $H \leftarrow H \setminus \{v\}$
        $L \leftarrow L \cup \{v\}$
        add v to Q

Chapter 5

Theoretical Guarantees

In this section, we will prove the correctness and analyze the running time of Algorithm 2.

5.1 Correctness

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 such that the following conditions hold for some d_1 and d_2:

Conditions.

(1) each node $v \in H$ is always moved to L if $d_H(v) < d_2$

(2) a node $v \in H$ is not moved to L if $d_H(v) \geq d_1$

Lemma 1 enables us to regard each sample as an independent uniformly random coin flip, whose outcome is heads if the node v samples a neighbour u which is in H, and 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 5 of Algorithm 3, the neighbour 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 5 of Algorithm 3 are mutually independent.

Proof. Consider H at a specific time. Let E_i be the event that v's i^{th} sample (Algorithm 3 Line 5) is in H. The event that any one neighbour 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 neighbour in H is exactly the number of its neighbours in H, $d_H(v)$, over its degree. □

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

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

Proof. For each node $v \in H$, $t(v)$ is initialized to the total number of samples, $k(v)$. Whenever v discovers that a sampled neighbour is not in H, either immediately in Line 6 of Algorithm 3 or later in Line 4 of Algorithm 4, $t(v)$ is decremented. So, $t(v) = k(v) -$ the number of sampled neighbours in L. □

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

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

Next, we relate $d_H(v)$ to the core values of nodes in L and H.

Lemma 4. *Assuming conditions (1) and (2) above hold, at Line 11 of Algorithm 2, 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) above, every node in H has $\geq d_2$ neighbours 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 $\geq d_1$ neighbours 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 neighbours 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). □

Now, we will prove that the bound $\frac{l \cdot k(v)}{deg(v)}$ (Algorithm 3 Line 8 and Algorithm 4 Line 6), 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 5. *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 $t(v) < \frac{l \cdot k(v)}{deg(v)}$

with high probability, 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 neighbour is in H. We will use the following Chernoff bounds from Theorem 2.1 in [1]. Let X be the sum of the number of heads and let $\mu = E[X]$. For $0 < \delta_1, \delta_2$

$$\Pr(X < (1 - \delta_1)\mu') \leq e^{\frac{-\delta_1^2 \mu'}{2}} \text{ for all } \mu' \leq \mu \tag{5.1}$$

and

$$\Pr(X \geq (1 + \delta_2)\mu') \leq e^{\frac{-\delta_2^2 \mu'}{2+\delta_2}} \text{ for all } \mu' \geq \mu \tag{5.2}$$

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 5.1, 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 5.2, $\Pr(t(v)) \geq l \cdot k(v)/deg(v)) = \Pr(t(v) \geq (1+\delta_2)\mu') \leq e^{\frac{-\mu'\delta_2^2}{2+\delta_2}} = e^{\frac{-\delta_2^2 l \cdot k(v)/deg(v)}{(1+\delta_2)(2+\delta_2)}}$.

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. □

Next, we show that Algorithm 2 satisfies the assumptions of Lemma 4 with high probability.

Lemma 6. *In any iteration of Algorithm 2,*

(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 $d_H(v) \geq l \cdot (1+\epsilon_1)$

with high probability.

Proof. Let $d_1 = l \cdot (1+\epsilon_1)$ and $d_2 = l/(1+(3/2)\epsilon_1)$. By Lemma 5, with high probability, no node $v \in V$ with $d_H(v) \geq d_1$ 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. □

We can now prove the bounds on the approximate degeneracy output by Algorithm 2.

Lemma 7. *Let l be the output returned by Algorithm 2 on Line 16 in the case a value is returned. With high probability, 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 6, 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 4, 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 2 gives a $(1+\epsilon)$-approximation.

Lemma 8. *Given a graph G, Algorithm 2 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 5 of Algorithm 2) which Algorithm 2 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 neighbours of any node $v \in H$. Let $k_i(v)$ and p_i be the number of neighbours sampled and the value of p, respectively, in any iteration i of the algorithm. The number of

neighbours of v which are sampled over the algorithm is:

$$
\begin{aligned}
&\leq \sum_{i=1}^{j} k_i(v) = \sum_{i=1}^{j} \lceil p_i deg(v) \rceil \\
&\leq \sum_{i=1}^{j} (p_i deg(v) + 1) = \log_{1+\epsilon_1}(n/l) + \sum_{i=1}^{j} p_i deg(v) \\
&= \log_{1+\epsilon_1}(n/l) + \frac{2((1+c)\ln(n)+\ln(\log_{1+\epsilon_1} n))(1+\epsilon_1)^2}{\epsilon_1^2 n} deg(v) \sum_{i=1}^{j} (1+\epsilon_1)^i \\
&\leq \log_{1+\epsilon_1}(n/l) + \frac{2((1+c)\ln(n)+\ln(\log_{1+\epsilon_1} n))(1+\epsilon_1)^2}{\epsilon_1^2 n} deg(v) \frac{(1+\epsilon_1)^{j+1}}{\epsilon_1} \\
&= \log_{1+\epsilon_1}(n/l) + \frac{2((1+c)\ln(n)+\ln(\log_{1+\epsilon_1} n))(1+\epsilon_1)^2}{\epsilon_1^2 n} deg(v) \frac{(1+\epsilon_1)(n/l)}{\epsilon_1} \\
&= \log_{1+\epsilon_1}(n/l) + \frac{2((1+c)\ln(n)+\ln(\log_{1+\epsilon_1} n))(1+\epsilon_1)^3}{\epsilon_1^3 l} deg(v)
\end{aligned}
$$

When sampling neighbours, every edge (u, v) may be examined when sampled in Line 5 of Algorithm 3 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 6, $d_H(v) < l \cdot (1+\epsilon_1)^2$ with high probability. 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 6, $d_H(u) < l \cdot (1+\epsilon_1)$. 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 cost of sampling neighbours 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 we rewrite as $\log_{1+\epsilon_1}(n/l) + \alpha(\ln(n) + \ln(\log_{1+\epsilon_1} n))$ for a constant α which depends on ϵ_1. 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 $2n\log_{1+\epsilon_1}(n/l) + 2\alpha n(\ln(n) + \ln(\log_{1+\epsilon_1} n)) = O(n \log n)$.

Lines 4 to 13 of Algorithm 3 samples neighbours from nodes in H and performs a constant amount of work per sampled neighbour. In particular we note that in the adjacency list model, each execution of Line 5 of Algorithm 3 requires only constant time. Lines 1 to 9 of Algorithm 4, 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 2 (Lines 5 to 16) is proportional to the number of samples.

To this we need to add the cost of the peeling algorithm at the end if no l is returned.

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 $O(\log n)$ with high probability. 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 17 of Algorithm 2 is $O(n \log n)$ with high probability. Therefore, we have the following lemma.

Lemma 11. *The total running time of Algorithm 2 is $O(n \log n)$.*

5.2.1 Running Time Comparison to the Colton Algorithm

To compare the running time of our algorithm against the algorithm from Colton et al. [18] (adapted for the adjacency list model as described in Section 3.3), we analyze the running time of Algorithm 2 with respect to the nodes that are sampled from.

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

Proof. By Lemma 7, the output l returned by Algorithm 2 is $> \delta/(1+\epsilon_1)^2$ with high probability, which means only nodes in N_s with degree $> \delta/(1+\epsilon_1)^2$ will be added to H and have their neighbours sampled. The number of iterations of the while-loop at Line 5 is $< \log_{1+\epsilon_1}(\frac{n(1+\epsilon_1)^2}{\delta})$. Let p denote the initial probability

$\frac{2((1+c)\ln(n)+\ln(\log_{1+\epsilon_1}(n)))(1+\epsilon_1)^2}{\epsilon_1^2 n}$. The total number of neighbours sampled is

$$
\begin{aligned}
&< \sum_{i=0}^{\log_{1+\epsilon_1}(\frac{n(1+\epsilon_1)^2}{\delta})} \sum_{v \in N_s} p(1+\epsilon_1)^i \cdot deg(v) \\
&= \sum_{i=0}^{\log_{1+\epsilon_1}(\frac{n(1+\epsilon_1)^2}{\delta})} p(1+\epsilon_1)^i \sum_{v \in N_s} deg(v) = p \cdot D_s \sum_{i=0}^{\log_{1+\epsilon_1}(\frac{n(1+\epsilon_1)^2}{\delta})} (1+\epsilon_1)^i \\
&= p \cdot D_s \left(\frac{(1+\epsilon_1)^{(\log_{1+\epsilon_1}(\frac{n(1+\epsilon_1)^2}{\delta})+1)} - 1}{(1+\epsilon_1)-1} \right) = p \cdot D_s \left(\frac{\frac{n(1+\epsilon_1)^3}{\delta} - 1}{(1+\epsilon_1)-1} \right) \\
&< p \cdot D_s \frac{n(1+\epsilon_1)^3}{\epsilon_1 \delta} = D_s \frac{2((1+c)\ln(n)+\ln(\log_{1+\epsilon_1}(n)))(1+\epsilon_1)^2}{\epsilon_1^2 n} \frac{n(1+\epsilon_1)^3}{\epsilon_1 \delta} \\
&= O(\tfrac{D_s}{\delta} \log n)
\end{aligned}
$$

Since Algorithm 2 runs proportionally to the number of samples, the running time is $O(\frac{D_s}{\delta} \log n)$. □

We use Lemma 12 to show that Algorithm 2 can have an asymptotically faster running time than the Colton algorithm adapted for the adjacency list model, which always samples $\Theta(n \log n)$ edges which results in 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 2 only samples neighbours 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 2 is $O(n^b \log n)$, which is asymptotically less than the running time of the Colton algorithm.

Chapter 6

Optimizations

In this chapter, we present optimizations to Algorithm 2 in order to improve its running time 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 end up removing 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. Then, we know that 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 to output a degeneracy 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 11 of Algorithm 2. Then, we binary search in the range $(1+\epsilon_1)^{2^{i-1}}$ to $(1+\epsilon_1)^{2^i}$ to get the first k where the factor $(1+\epsilon_1)^k$ results in H not being empty.

6.3 Reusing Randomness

On Line 5 of Algorithm 3, computing a random neighbour 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 5, 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} neighbour of a node v (Algorithm 3 Line 5), 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 neighbours for v from the previous iteration and add to it as needed.

Chapter 7

Extension to k-core Decomposition

We can also extend Algorithm 2 to solve full k-core decomposition. In this section, we describe how we 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 with high probability in $O(n \log n)$ time.

7.1 Algorithm Modifications

Instead of returning l as the approximate degeneracy when H is non-empty on Line 16 of Algorithm 2, we continue the algorithm and assign an approximate core number label l to nodes which remain in H (Algorithm 2 Line 15). Within the body of the else statement in Algorithm 2 on Line 15, 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 17 of Algorithm 2, when $p \geq 1$, we run the peeling algorithm on the remaining unlabelled nodes V and assign core numbers to nodes which were not labeled inside the while-loop. 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.

7.2 Correctness

The high probability of correctness for the approximate core values follow from the Lemmas 1 to 8 in Section 5.2. However, we need to prove that using the peeling algorithm as described for labelling the unlabeled nodes satisfies the desired approximation bounds. This is a concern because nodes with lower core values may have already been labelled and removed from the graph of unlabelled nodes, and therefore will not participate in the peeling algorithm. Consequently it is possible that nodes in the peeling algorithm will be assigned a higher core value than their actual core value.

Lemma 13. *Running the peeling algorithm on the unlabelled nodes (as described above) assigns* $(1 + \epsilon)$*-approximate core numbers to those nodes.*

Proof. Let V' be the set of unlabelled nodes and let l' be the last label assigned within the while-loop (described above). Consider the nodes in V' which are assigned a label of $< l'/(1 + (3/2)\epsilon_1)$ by running the peeling algorithm on V'. By Lemma 7, all nodes with core value of $< l'/(1+(3/2)\epsilon_1)$ would also be unlabelled. Hence, the nodes which are assigned a label of $< l'/(1 + (3/2)\epsilon_1)$ by the peeling algorithm receive the same label as they would in the peeling algorithm run on the entire graph.

Next, consider the nodes $v \in V'$ which would have received a label of $> l'/(1 + (3/2)\epsilon_1)$ by running the peeling algorithm on V', but we label them by l'. By the proof of Lemma 7, since v was moved to L in the last iteration, $c(v) < l' \cdot (1 + \epsilon_1)^2$. Hence, the approximation bounds hold if we label them with l'. □

7.3 Running Time

We slightly modify the analysis of the running time in Section 5.2 to consider the time it takes to label all nodes with an approximate core number. Let $l(v)$ denote the label assigned to any node v. The neighbours of v are sampled in each iteration until does not get moved to L and gets assigned a label. The maximum number of iterations of the while-loop is $j = \log_{1+\epsilon_1} n$ which means the total cost of sampling node v is $\log_{1+\epsilon_1}(n) + \frac{2((1+c)\ln(n)+\ln(\log_{1+\epsilon_1} n))(1+\epsilon_1)^3}{\epsilon_1^3} \deg(v)$.

Similar to Section 5.2, consider the iteration where node v is labelled $l(v)$ (v was moved to L in the previous iteration). In the previous iteration, for each edge (u, v), if u was moved to L before v, we attribute the cost of sampling that edge in iteration j to u. Otherwise, we attribute that edge to v. This time, we consider the edges attributed to the nodes which are assigned a label in each iteration, and by the same analysis as in Section 5.2, the edges attributed for nodes with labels $l(v)$ in any iteration is $< l(v)(1+\epsilon)^2$. So, the number of samples of attributed edges for any node with label $l(v)$ in any iteration is less than $\log_{1+\epsilon_1}(n)+\alpha(\ln(n)+\ln(\log_{1+\epsilon_1} n))$ for some constant α. Since this quantity is the same for all labels, and each node is assigned one label, this bound when summed over all nodes is $n\log_{1+\epsilon_1}(n) + \alpha n(\ln(n) + \ln(\log_{1+\epsilon_1} n))$. Then, the total cost of sampling is bound by twice this which is $2n\log_{1+\epsilon_1}(n) + 2\alpha n(\ln(n) + \ln(\log_{1+\epsilon_1} n)) = O(n \log n)$.

Chapter 8

Experiments

We refer to our approach as NESD (Neighbour Sampling for Degeneracy). In our experiments, we compare NESD (including all the optimizations from Chapter 6) to the approximation algorithm from [18] and the state-of-the-art exact degeneracy algorithm, SemiDeg+, from [22]. Since the goal of our approach is to achieve an efficient sublinear $(1+\epsilon)$ approximation algorithm, we do not compare versus other less accurate approximate algorithms nor other exact algorithms since they are superseded by SemiDeg+. We experiment with 5 different values of ϵ, namely 0.5, 0.25, 0.1, 0.05, 0.01 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 use the datasets twitter-2010, sk-2005, gsh-2015, uk-2014, and clueweb12 from

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 8.1: Summary of datasets

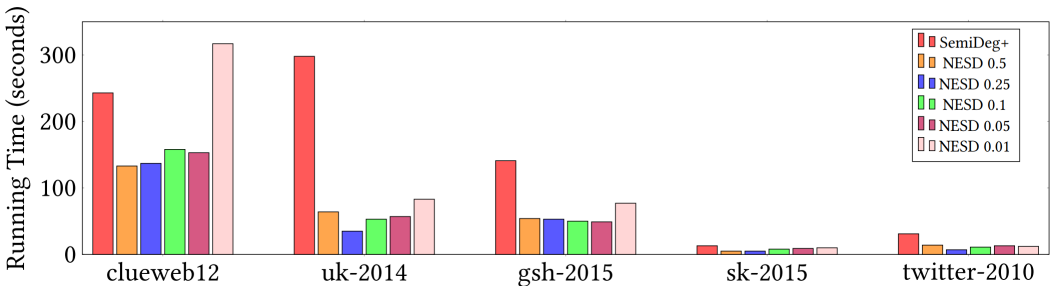

Figure 8.1: Running time comparison between SemiDeg+ and NESD with various ϵ and $c = 0.5$

the Laboratory of Web Algorithmics [8, 7, 6], and the characteristic of each graph is illustrated in Table 8.1. We note that the number of edges shown in Table 8.1 is after symmetrization, where we add the reverse of directed 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 8.1 shows the running time results of NESD with each ϵ. On each of the datasets, the approximation algorithm from [18] 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 2.17x to 4.56x faster than SemiDeg+ on twitter-2010, 1.33x to 2.61x faster on sk-2005, 1.81x to 2.86x faster on gsh-2015, and 3.57x to 8.45x faster on uk-2014. On clueweb12, NESD ran 1.53x to 1.83x faster with ϵ between 0.5 to 0.05, but 1.3x slower on $\epsilon = 0.01$, which is the only input in all the experiments where NESD was slower.

Figure 8.2 shows the degeneracy δ of each dataset (output by SemiDeg+) as well as the approximate degeneracy and error factors from the output by NESD with each

	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
NESD 0.01	4238.685	0.999	9998.635	0.999	9645.45	0.975	4502.467	0.998	2534.374	1.019

Figure 8.2: Error factors of NESD degeneracy outputs with various ϵ

ϵ. 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 $\epsilon = 0.1$, between 0.975 and 1.023 for $\epsilon = 0.05$, and between 0.975 and 1.019 for $\epsilon = 0.01$. 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.

Chapter 9

Conclusion and Future Work

We have devised $O(n \log n)$ algorithms for degeneracy and for k-core decomposition in the adjacency list model. Both algorithms give provably close approximate solutions, with high probability. We have shown in experiments, that on all our instances of very large graphs, our algorithm for degeneracy works within our approximation guarantees, and for almost all our data, 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.

The natural next stage of our work is to extend it to the dynamic graph setting. From the existing related work, it seems clear that our algorithm would work for the streaming setting since we could sample each edge as they are added similar to the algorithm in [16]. However, it is less clear how to manage edge removals (with minimal update time and additional space) if that edge was previously sampled. Another natural extension of our algorithm would be to implement it in a parallel computing environment, for example where each node samples its neighbours in parallel, and compare the results to other parallel algorithm implementations.

Bibliography

[1] Anders Aaman, Adam Karczmarz, Jakub Łacki, Nikos Parotsidis, Peter M. R. Rasmussen, and Mikkel Thorup. Optimal decremental connectivity in non-sparse graphs, 2021.

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

[3] V Batagelj and M Zaversnik. An o(m) algorithm for cores decomposition of networks. 2003.

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

[5] Sayan Bhattacharya, Monika Henzinger, Danupon Nanongkai, and Charalampos Tsourakakis. Space- and time-efficient algorithm for maintaining dense subgraphs on one-pass dynamic streams. STOC '15, page 173–182, New York, NY, USA, 2015. Association for Computing Machinery.

[6] 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.

[7] 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.

[8] 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.

[9] Bela. Bollobas. *Extremal graph theory*. Dover Publications, Mineola, N.Y, dover edition. edition, 2004 - 1978.

[10] Moses Charikar. Greedy approximation algorithms for finding dense components in a graph. In Klaus Jansen and Samir Khuller, editors, *Approximation Algorithms for Combinatorial Optimization*, pages 84–95, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[11] 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.

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

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

[14] David Eppstein. Arboricity and bipartite subgraph listing algorithms. *Information processing letters*, 51(4):207–211, 1994.

[15] 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.

[16] 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.

[17] Martin Farach-Colton and Meng-Tsung Tsai. Computing the degeneracy of large graphs. 03 2014.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] Min Chih Lin, Francisco J. Soulignac, and Jayme Luiz Szwarcfiter. Arboricity, h-index, and dynamic algorithms. *CoRR*, abs/1005.2211, 2010.

[25] David W. Matula and Leland L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *J. ACM*, 30(3):417–427, jul 1983.

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

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

[28] 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.