Efficient Computation of Probabilistic Core Decomposition at Web-Scale

Fatemeh Esfahani, Venkatesh Srinivasan, Alex Thomo, and Kui Wu
Computer Science Dept., University of Victoria, B.C., Canada
{esfahani,srinivas,thomo,wkui}@uvic.ca

ABSTRACT

Core decomposition is a popular tool for analyzing the structure of network graphs. For probabilistic graphs the computation comes with several challenges and the state-of-the-art approach is not scalable to large graphs. One of the challenges is to compute tail probabilities of vertex degrees in probabilistic graphs. To address this we employ a special version of the Central Limit Theorem (CLT) to obtain the tail probabilities efficiently. Based on our CLT methodology we propose a peeling algorithm to compute the core decomposition of a probabilistic graph that scales to very large graphs and is orders of magnitude faster than the state-of-the-art. Next, we propose a second algorithm that can handle graphs not fitting in memory by processing them sequentially one vertex at a time. This algorithm has the desirable property that it can produce close approximations to true core numbers of vertices in only a fraction of iterations needed for full completion. The graphs in our study are orders of magnitude larger than those considered in the literature. Our extensive experimental results confirm the scalability and efficiency of our algorithms; the largest graph we can process has more than 40 million nodes and 1.5 billion edges and we are able to compute its core decomposition on a commodity machine in about two and half hours.

1 INTRODUCTION

Probabilistic graphs are graphs in which each edge has a probability of existence (cf. [6–8, 19, 21, 43]). Mining probabilistic graphs has become the focus of interest in analyzing many real-world datasets, such as social, trust, communication, and biological networks due to the intrinsic uncertainty present in them. For instance, influence between users (cf. [7, 17, 21]) in a social network can be modeled as a probabilistic graph with probabilities on the edges representing the likelihood that some action of one user will be adopted by another. In terms of trust inference, probabilistic models with trust values as edge probabilities can be used to compute trust associated with a social relationship [27, 28]. In protein-protein networks (cf. [9]) interactions between proteins are obtained through laboratory experiments that are prone to measurement errors resulting in edges labeled with confidence levels that can also be interpreted as probabilities [14, 15, 41].

Discovering dense components is of great importance in analyzing network graphs [29]. A popular way to find such components is core decomposition which has been shown to have a wide variety of applications (cf. [1, 24, 30, 44, 46]). For instance, it can be used in measuring structural diversity is social contagion [44], describing biological functions of proteins in protein-protein interaction networks [30], analyzing network structure's properties to explore collaboration in software teams [46], and also as a metric for sentence selection in text summarization [1].

Probabilistic core decomposition naturally extends all the applications of deterministic core decomposition to probabilistic graphs. Other applications, showcased in [6], are facilitating influence maximization and task-driven team formation in probabilistic graphs.

In k-core computation the goal is to find the maximal subgraph in which each vertex has at least k neighbors within that subgraph. The set of all k-cores of a graph, for various k, forms the core decomposition of that graph [40]. Core decomposition in deterministic graphs has been thoroughly studied in literature [3, 11, 24, 35], and can be computed in $O(m)$ time, where m is the number of the edges in the input graph. However, in the probabilistic context, computing core decomposition is much more challenging.

Here we use the probabilistic (k, η)-core notion introduced by Bonchi, Gullo, Kaltenbrunner, and Volkovich in [6]. In (k, η)-core computation the goal is to find the maximal subgraph in which each vertex has at least k neighbors within that subgraph with probability no less than $\eta \in [0, 1]$. Threshold η is given by the user and defines the desired level of certainty of the output cores. A fundamental notion needed to compute the (k, η)-core is the η-degree of a vertex v. It is the maximum degree such that the probability for v to have that degree is no less than η.

Challenges and contributions. A significant initial challenge is computing η-degrees of graph vertices. In [6], the η-degree of each vertex v is computed using dynamic programming (DP) which has a complexity of $O(d_v^2)$, where d_v is the number of edges incident to v. Unfortunately, in many real social and web networks, d_v can be in the millions and a quadratic algorithm such as DP is impractical.

Our first contribution is the design of an efficient method for computing η-degrees. Our method is based on Lyapunov's special version of the Central Limit Theorem [25, 34] and we show its output to be virtually indistinguishable from exact computation for vertices with a high number of incident edges.

While solving the challenge of computing η-degrees is an important step forward, we still need efficient algorithms to compute core decomposition for large probabilistic graphs. We propose two efficient algorithms to solve this problem.

The first one, which we call the "peeling algorithm" (PA), recursively deletes (peels-off) the vertex of the smallest degree. Our contribution here is in designing efficient arrays for storing important bookkeeping information. Handling these arrays becomes challenging because, differently from the deterministic case, the process of keeping the vertices sorted based on their changing η-degrees is more complicated and we need to shuffle information carefully in order to keep the arrays up to date. Notably, our PA algorithm scales to datasets two orders of magnitude bigger than those that the state-of-the-art algorithm [6] can handle.

For the case when the input graph does not fit in memory, we propose a sequential algorithm (SA) based on the vertex-centric model of computation. The main idea of the SA algorithm is to

maintain an upper-bound, called vertex value, on the core number of each vertex. This upper-bound is initialized to be the η-degree of each vertex, and after each iteration of the algorithm it is tightened further until it reaches the exact core value. While being moderately slower than PA, there are two notable advantages associated with this algorithm. First, the SA algorithm has a memory footprint of $O(n)$ as opposed to $O(m)$ for PA, where n and m are the number of vertices and edges, respectively. This amounts to SA requiring 30 to 40 times lower memory for social and web network graphs in practice. Second, as shown in Section 6, after only a fraction of iterations of the algorithm, we can obtain an approximation very close to the true core numbers of vertices.

In summary, our contributions are as follows.

- We introduce an efficient approach to compute η-degrees using Lyapunov's central limit theorem which gives very accurate approximations on the probability that a vertex can have a certain degree. We prove the accuracy of the approach and show that this method of computing probabilistic degree is numerically stable.
- We propose a peeling algorithm (PA) based on recursive vertex deletions which, by using carefully engineered array structures, is able to scale to graphs two orders of magnitude larger than what the state-of-the-art algorithm can handle.
- For the case when the input graph does not fit into memory, we propose a sequential algorithm (SA) to produce the core decomposition in probabilistic graphs with a low memory footprint. This algorithm can also produce accurate approximations after only a fraction of total iterations, a useful feature to have in applications when exact core numbers are not necessary.

2 BACKGROUND

Cores of deterministic graphs. Let $G = (V, E)$ be an undirected graph, where V is a set of n vertices, and E is a set of m edges. For vertex $v \in V$, let $N_G(v)$ be the set of v's neighbors: $N_G(v) = \{u : (u, v) \in E\}$. The (deterministic) degree of v in G, is equal to $|N_G(v)|$.

Given $V' \subseteq V$, and $E_{V'} = \{(u, v) \in E : u, v \in V'\}$, graph $H = (V', E_{V'})$ is called the subgraph of G induced by V'. Let $k \in [0, d_{\max}(G)]$, where $d_{\max}(G)$ is the maximum vertex degree in G. The k-core of G is defined as the maximal induced subgraph $C_k(G) = (V', E_{V'})$ in which each vertex $v \in V'$ has degree of at least k. The set of all k-cores forms the core decomposition of G. Core decomposition of G is unique and it satisfies the following relation [40]: $G = C_0(G) \supseteq \cdots \supseteq C_{d_{\max}(G)-1} \supseteq C_{d_{\max}(G)}$. The coreness (or core number) of a vertex is defined as the maximum value of k such that the corresponding $C_k(G)$ contains v.

Probabilistic graphs. A probabilistic graph is a triple $\mathcal{G} = (V, E, p)$, where V and E are as before and $p : E \to (0, 1]$ is a function that maps each edge $e \in E$ to its existence probability p_e. For each vertex $v \in V$, the set of edges incident to v is denoted by $\mathcal{N}_v$. $d_v = |\mathcal{N}_v|$ is the number of all edges incident to v which is equal to the deterministic degree of v. In the most common probabilistic graph model (cf. [6, 19, 21]), the existence probability of each edge is assumed to be independent of other edges.

In order to analyze probabilistic graphs, we use the concept of *possible worlds* that are deterministic graph instances of $\mathcal{G}$ in which only a subset of edges appears. The probability of a possible world $G \sqsubseteq \mathcal{G}$ is obtained as follows: $\Pr(G) = \prod_{e \in E_G} p_e \prod_{e \in E \setminus E_G} (1 - p_e)$.

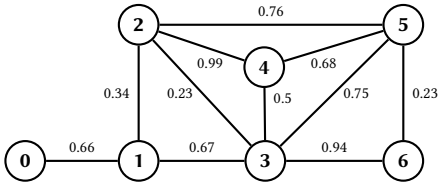

Figure 1: A probabilistic graph.

In a probabilistic graph $\mathcal{G}$, the notion of *η-degree*, where $\eta \in [0, 1]$, denoted by *η-deg*(v), of a vertex v is defined in [6] as the maximum k for which $\Pr_{G \sqsubseteq \mathcal{G}}[d_G(v) \geq k] \geq \eta$, where $k = 0, \ldots, d_v$, and the probability is taken over all the possible worlds $G \sqsubseteq \mathcal{G}$.

For instance, consider Figure 1. When $\eta = 0.5$, vertex 6 has degree at least 2 with probability $0.94 \cdot 0.23 = 0.2162$ (product of probabilities that each of the two edges is in a possible world), and it has degree at least 1 with the probability $1 - ((1 - 0.94) \cdot (1 - 0.23)) = 0.9538$ (complementary probability that none of the two edges is in a possible world) which is greater than the threshold. Thus, the η-degree of vertex 6 is 1.

In the rest of the paper, we use $\Pr[d(v) \geq k]$ to denote $\Pr_{G \sqsubseteq \mathcal{G}}[d_G(v) \geq k]$. The value of $\Pr[d(v) \geq k]$ decreases with the increase of k. Note that d_v is different from $d(v)$; the former is constant, whereas the latter is a random variable.

Cores of probabilistic graphs. In order to extend k-core decomposition to probabilistic graphs, the notion of (k, η)-*core* is defined in [6]:

Definition 1. *Given a probabilistic graph $\mathcal{G} = (V, E, p)$, and a threshold $\eta \in [0, 1]$, (k, η)-core is the maximal induced subgraph $C_{(k,\eta)}(\mathcal{G}) = (V', E_{V'}, p)$ in which the η-degree of each vertex $v \in V'$ is at least k. The set of all (k, η)-cores forms the core decomposition of $\mathcal{G}$.*

The core decomposition in probabilistic graphs is unique, and the (k, η)-cores are nested into each other similar to the deterministic case. The highest value of k for which v belongs to a (k, η)-core is called *η-core* number or probabilistic coreness of v.

Computing η-degrees using Dynamic Programming. We have that $\Pr[d(v) \geq k] = 1 - \sum_{i=0}^{k-1} \Pr[d(v) = i]$. One way of computing $\Pr[d(v) = i]$ is to use dynamic programming as proposed in [6]. However, this method of computing the η-degree has complexity of $O(d_v^2)$ for a vertex v of deterministic degree d_v. This is not practical when d_v is big, say over 20 thousand, which occurs often in all our datasets. In fact, DP cannot finish in reasonable time even for one such vertex. In addition, web-scale graphs normally have millions of nodes with moderate-high degree (e.g., a thousand or more), and if DP is applied to every such node, the total processing time increases considerably. In the next section we introduce an alternative way for fast computation of the η-degree of a vertex v using Lyapunov central limit theorem.

3 COMPUTING η-DEGREES USING CENTRAL LIMIT THEOREM

In this section, we first show how a special version of Central Limit Theorem (CLT) can be applied to accurately estimate $\Pr\big[d(v) \geq k\big]$. Then, we show theoretical bounds on the accuracy of this approximation. Specifically, we show that CLT can produce a very accurate approximation to tail probabilities of the vertex degree.

CLT, one of the most important theorems in statistics, states that given a set of random variables, their properly scaled sum converges to a normal distribution under certain conditions. There are different versions of CLT, with the most common one focusing on independent identically distributed (i.i.d.) random variables. In this paper, we consider a variant called Lyapunov CLT [33, 34] that can be applied when random variables are independent, but not necessarily identically distributed. The Lyapunov's condition imposes a limit on the growth of the third absolute central moment of each random variable in the given sequence ensuring the convergence of the normalized sum of that sequence to standard normal distribution. Formally, we have

Theorem 3.1. ***Lyapunov CLT [25].*** *Let $\xi_1, \xi_2, \cdots, \xi_n$ be a sequence of independent, but non-identically distributed random variables, each with finite expected value μ_k and variance σ_k. Let*

$$s_n^2 = \sum_{k=1}^{n} \sigma_k^2, \tag{1}$$

Lyapunov CLT states that if

$$\lim_{n\to\infty} \frac{1}{s_n^{2+\delta}} \sum_{k=1}^{n} \mathrm{E}[|\xi_k - \mu_k|^{2+\delta}] = 0, \tag{2}$$

for some $\delta > 0$, then $\frac{1}{s_n}\sum_{k=1}^{n}(\xi_k - \mu_k)$ converges in distribution to a standard normal random variable.

Equation (2) is called Lyapunov's condition which in practice is usually tested for the special case $\delta = 1$. The proof for this theorem can be found in [4, 13].

Computing η-degrees using Lyapunov CLT. In what follows, we show how Lyapunov CLT can help compute $\Pr\big[d(v) \geq k\big]$, for each vertex v of the input probabilistic graph $\mathcal{G}$.

Recall that for each vertex v we have a set of edges incident to v denoted by $\mathcal{N}_v$. Each e_i in $\mathcal{N}_v$ has existence probability p_i, which is independent of the other edge probabilities in $\mathcal{G}$. Corresponding to each edge e_i in $\mathcal{N}_v$, we define a Bernoulli random variable X_i which takes on 1 with probability p_i, and 0 with probability $(1 - p_i)$. Formally,

$$X_i = \begin{cases} 1 & \text{if edge } e_i \text{ incident to } v \text{ exists in the graph} \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

Since X_i is a Bernoulli random variable, we know that $\mathrm{E}[X_i] = \mu_i = p_i$ and $\mathrm{Var}[X_i] = p_i(1 - p_i)$. Using the fact that $d(v) = \sum_{i=1}^{d_v} X_i$, we have:

$$\Pr[d(v) \geq k] = \Pr\left[\sum_{i=1}^{d_v} X_i \geq k\right]. \tag{4}$$

According to Equation (4), finding the probability that a vertex v is of the degree at least k is equivalent to computing the probability that the sum of X_i's for v is at least k. In addition, Bernoulli random variables X_i's are independent, but not identically distributed. Thus, if condition (2) is satisfied and if d_v is large enough, we can conclude that $\frac{1}{s_{d_v}}\sum_{i=1}^{d_v}(X_i - \mu_i)$ has standard normal distribution, where $s_{d_v} = \sqrt{\sum_{i=1}^{d_v} p_i(1-p_i)}$. To compute $\Pr[\sum_{i=1}^{d_v} X_i \geq k]$, we can subtract $\sum_{i=1}^{d_v}\mu_i$ from both sides of the inequality, and then divide by s_{d_v} which results in:

$$\Pr\Big[\sum_{i=1}^{d_v} X_i \geq k\Big] = \Pr\Big[\frac{1}{s_{d_v}}\sum_{i=1}^{d_v}(X_i - \mu_i) \geq \frac{1}{s_{d_v}}(k - \sum_{i=1}^{d_v}\mu_i)\Big]. \tag{5}$$

Using Lyapunov CLT, and setting

$$Z = \frac{1}{s_{d_v}}\sum_{i=1}^{d_v}(X_i - \mu_i), \tag{6}$$

we can say that Z has standard normal distribution. Thus

$$\Pr\big[d(v) \geq k\big] \cong \Pr\big[Z \geq z\big], \tag{7}$$

where $z = \frac{1}{s_{d_v}}(k - \sum_{i=1}^{d_v}\mu_i)$. In fact, since Z in Equation (6) has standard normal distribution, using the complementary cumulative distribution function [56], we can efficiently evaluate the right hand side of Equation (7). To find the η-degree we start with $k = 1$, and approximate $\Pr[d(v) \geq k]$ using Lyapunov CLT, finding the maximum k for which the probability is above threshold η.

We can apply Theorem 3.1 provided that Lyapunov's condition is satisfied. By setting $\delta = 1$ in Equation (2) we show that this condition holds for a sequence of non-identically distributed Bernoulli random variables.

Theorem 3.2. *Given a sequence of random variables $X_i \sim$ Bernoulli(p_i), for $i \in [1, n]$, the Lyapunov's condition (2) for $\delta = 1$ is satisfied whenever $s_n^2 = \sum_{k=1}^{n} p_k(1 - p_k) \to \infty$.*

Proof. For each Bernoulli random variable X_i we know that $\sigma_i^2 = p_i(1 - p_i)$, and $\mu_i = p_i$. Therefore, $s_n^2 = \sum_{k=1}^{n} p_k(1 - p_k)$ according to the equation (1). On the other hand, when $\delta = 1$, $\mathrm{E}[|X_k - \mu_k|^3]$ is computed as follows:

$$\begin{aligned} \mathrm{E}[|X_k - \mu_k|^3] &= p_k(1-p_k)^3 + (1-p_k)p_k^3, \\ &= p_k(1-p_k)[(1-p_k)^2 + p_k^2] \leq \sigma_k^2, \end{aligned} \tag{8}$$

where in inequality (8) we have used the fact that $(1-p_k)^2 + p_k^2 \leq 1$. Thus, $\sum_{k=1}^{n}\mathrm{E}[|X_k - \mu_k|^3] \leq s_n^2$. Substituting this in the Lyapunov's condition (2) for $\delta = 1$, we conclude that the condition is satisfied whenever $s_n^2/s_n^3 \to 0$ as $n \to \infty$ which means that s_n should go to infinity as n increases. This is equivalent to $s_n^2 = \sum_{k=1}^{n} p_k(1 - p_k) \to \infty$, and as a result the theorem follows. In other words, since we have

$$s_n^2 = \sum_{k=1}^{n} p_k(1 - p_k) = \sum_{k=1}^{n} \sigma_k^2 \geq n\sigma_{\min}^2, \tag{9}$$

for large n and as long as $\sigma_{\min}^2 = \min_k\{\sigma_k^2\}$ is away from zero, $n\sigma_{\min}^2 \to \infty$ which results in $s_n^2 \to \infty$ as n approaches infinity. □

Accuracy of the Approximation. In order to show the accuracy of the approximation, we refer to the Berry–Esseen theorem [57]: For a given sequence $Y_1, Y_2, \ldots$ of non identically distributed and independent random variables with $E(Y_i) = 0$, $E(Y_i^2) = \sigma_i^2$, and $E(|Y_i^3|) = \rho_i < \infty$, there exists a constant C_0 such that the following inequality is satisfied for all n:

$$\sup_{x\in\mathbb{R}} |F_n(x) - \Phi(x)| \leq C_0 \cdot \psi_0, \tag{10}$$

where F_n is the cumulative distribution of $S_n = \frac{Y_1 + Y_2 + \cdots + Y_n}{\sqrt{\sigma_1^2 + \sigma_2^2 + \cdots + \sigma_n^2}}$, which is the sum of Y_i's standardized by the variances, and Φ is the cumulative distribution of the standard normal distribution. In the above inequality ψ_0 is a function given by

$$\psi_0 = \psi_0(\overrightarrow{\sigma}, \overrightarrow{\rho}) = \left(\sum_{i=1}^{n}\sigma_i^2\right)^{-3/2} \cdot \sum_{i=1}^{n}\rho_i. \tag{11}$$

where $\overrightarrow{\sigma} = (\sigma_1, \cdots, \sigma_n)$, and $\overrightarrow{\rho} = (\rho_1, \cdots, \rho_n)$ are the vectors

of σ_i's and ρ_i's respectively. It should be noted that the best upper-bound obtained so far for C_0 is 0.56 [42].

Based on the Berry–Esseen theorem, the more the number of edges incident to a vertex, the better the accuracy of the results. In the following corollary, we show how to obtain an upper-bound on the maximal error while approximating the true distribution of the sum of X_i's with the normal distribution.

Corollary 1. *For each vertex v in the probabilistic graph $\mathcal{G}$ with X_i's being Bernoulli random variables as defined in (3), where $i = 1, \ldots, d_v$, the error bound on the approximation of the right-hand side of Equation (7) to the standard normal distribution is given as follows:*

$$\sup_{x \in \mathbb{R}} \left| F_{d_v}(x) - \Phi(x) \right| \leq \frac{0.56}{\sqrt{p_1(1-p_1) + \cdots + p_{d_v}(1-p_{d_v})}}$$

Proof. Setting $Y_i = X_i - p_i$ in equation (6), we can apply the Berry–Essen theorem for random variables $Y_1, Y_2, \cdots, Y_{d_v}$, since for each Y_i, $\mathrm{E}[Y_i] = 0$. In addition,

$$\begin{aligned} \mathrm{E}[Y_i^2] &= \mathrm{E}[(X_i - p_i)^2] = \mathrm{Var}[X_i] = \sigma_i^2 = p_i(1-p_i), \\ \mathrm{E}[|Y_i^3|] &= \mathrm{E}[|X_i - p_i|^3] = \rho_i = (1-p_i)^3 p_i + p_i^3(1-p_i) \\ &= p_i(1-p_i)[(1-p_i)^2 + p_i^2] < \infty. \end{aligned} \tag{12}$$

It should be noted that the random variable

$$S_{d_v} = \frac{(X_1 - p_1) + (X_2 - p_2) + \cdots + (X_{d_v} - p_{d_v})}{\sqrt{\sigma_1^2 + \sigma_2^2 + \cdots + \sigma_{d_v}^2}} \tag{13}$$

in the Berry–Essen theorem is the same as the random variable Z in Equation (6).

Substituting σ_i^2 and ρ_i in Equation (11), we obtain:

$$\begin{aligned} \psi_0 &= \psi_0(\overrightarrow{\sigma}, \overrightarrow{\rho}) \\ &= \left(\sum_{i=1}^{d_v} p_i(1-p_i) \right)^{-3/2} \cdot \left(\sum_{i=1}^{d_v} p_i(1-p_i)[(1-p_i)^2 + p_i^2] \right), \end{aligned} \tag{14}$$

Using the fact that $1 = (1-p_i+p_i)^2 = (1-p_i)^2 + p_i^2 + 2p_i(1-p_i) \geq (1-p_i)^2 + p_i^2$, we can simplify (14) to have:

$$\begin{aligned} \psi_0 &\leq \left(\sum_{i=1}^{d_v} p_i(1-p_i) \right)^{-3/2} \cdot \left(\sum_{i=1}^{d_v} p_i(1-p_i) \right) \\ &= \left(\sum_{i=1}^{d_v} p_i(1-p_i) \right)^{-1/2} = \frac{1}{\sqrt{p_1(1-p_1) + \cdots + p_{d_v}(1-p_{d_v})}}, \end{aligned} \tag{15}$$

Substituting (15) in (10) the stated claim follows. □

4 PEELING ALGORITHM (PA)

In this section we propose a graph peeling algorithm (PA). Graph peeling, that is, (1) recursively deleting the vertex v of smallest degree (2) setting v's coreness to be equal to its degree at the time of deletion, and (3) updating the degrees of v's neighbors while keeping them sorted, is a general idea that has been used broadly in core decomposition of deterministic graphs (cf. [3, 24]). However, it requires substantial algorithmic engineering in order to achieve high scalability when applied to probabilistic graphs. This is because when a vertex v is deleted in the peeling process, updating the η-degrees of v's neighbors and maintaining the data structures up to date are non-trivial. In order to tackle these challenges, we use efficient *array structures* and *lazy updates*, which delay updating the η-degrees of v's neighbors for as long as possible.

Computing and updating η-degrees. An expensive step in the peel-off process is computing initial η-degrees and updating them for those vertices that lose neighbors in a peel-off step. We will depart from the feature of the deterministic case of having vertices sorted at all times based on their current degrees and allow instead the vertices to not be on their precise order as long as at the time of their removal we can fix things up using the key functions and data structures that we define.

More specifically, the computation and update of η-degrees is delayed as much as possible by using easy to compute lower-bounds on η-degrees instead of exact values. It is only when a vertex is candidate for removal that we compute its exact η-degree. At that time the remaining graph is typically much smaller and the computation becomes significantly faster.

Based on Corollary 1, we know that Lyapunov CLT gives a very good approximation on tail probabilities of vertex degrees. We use the values produced by CLT (decremented by a small epsilon) in order to obtain lower-bounds on η-degrees. For simplicity of exposition, we refer to the lower-bound values simply as *values*.

One important difference between the core decomposition of probabilistic and deterministic graphs is that the η-degree of a vertex can decrease by *at most one* when a neighbor is removed (according to Lemma 2 in [6]) as opposed to *exactly one* in deterministic graphs. An array **A** stores, for each value in the input graph, the set of vertices with that value. In the PA algorithm at each iteration, we decrease the value of a vertex v by one if a v's neighbor is removed. When v's turn comes to be removed and processed, we compute its η-degree and if it is the same as its current value we remove v. Otherwise, we repeatedly swap v to the proper place in **A**. There could be several neighbor removals that did not change the η-degree of v, thus the proper place of v in **A** could be far from the next block of vertices, and therefore we might need to perform several swaps.

PA algorithm description. The PA algorithm is given in Algorithm 1. There we have arrays **d**, **b**, **p**, and **A**. For an example see Figure 1 and Table 1. Vertices in the PA algorithm are assumed to be labeled by numbers 0 to $n-1$. Array **d** initially stores for each vertex the lower-bound on the η-degree of that vertex. For instance, vertex 1 has a lower-bound of 1 in Table 1, so $\mathbf{d}[1] = 1$. Initially, vertices are stored in **A** in ascending order of their lower-bounds. We have colored **A** in shades of green in Table 1. The first block in **A** contains all the vertices with a lower-bound equal to 0; the second block contains vertices with lower-bound equal to 1, and so on. In order to determine the index boundaries of such blocks in **A**, we define array **b** which stores the index boundaries of the vertex blocks in **A**. In Table 1 we have for instance $\mathbf{b}[1] = 2$ and $\mathbf{b}[2] = 6$. In order to handle swapping efficiently we define an array **p** which stores the position of each vertex in **A**. For instance, vertex 6 is at position 2 in **A**, therefore $\mathbf{p}[6] = 2$.

There are two additional arrays as well; **gone** and **valid**. Since we do not remove vertices physically from the graph, we use array **gone** to keep track of the removed vertices at each step of the algorithm. Array **valid** tells for each vertex v if the η-degree of v is the same as the value $\mathbf{d}[v]$. The array **gone** is initially set to false for all the vertices, because none of the vertices has been removed yet. Array **valid** is set to all-false vector because all the vertices are on their lower-bound at the beginning of the algorithm. For instance, in Table 1, we have $\mathbf{valid}[4] = false$

which indicates that vertex 4 is on its lower-bound and its η-degree should eventually be computed.

We illustrate a few steps of the PA algorithm on the graph in Figure 1. We set $\eta = 0.5$. The PA algorithm starts with the first vertex v in array **A**, checking whether v is on its lower-bound or its η-degree is available. As can be seen in Table 1, vertex 0 is the first vertex in **A**. Since **valid**[0] = $false$, the vertex 0 is on its lower-bound, and its η-degree might be higher. Therefore, it might not be its turn to be removed. As such, the compute_swap function is called (line 19, Algorithm 1) to compute the η-degree, and then do the required number of swaps to the right, using Algorithm 3, to place the vertex in the proper block of **A**. *Thus, unlike deterministic case we do need to perform several swaps to the right.* The compute_swap function (Algorithm 4) computes the η-degree of vertex 0, which turns out to be 1. Thus, the vertex only needs one swap to the right in **A** to be placed in the block containing all the vertices with degree 1. In order to do each swap in constant time (using Algorithm 3), we swap vertex 0 with the last vertex in the same block which is 6. As a result, the positions of vertices 0 and 6 should be swapped as well: **p**[0] = 2, **p**[6] = 1. Vertex 0 becomes the last element of its current block (the block of vertices with degree 0). In order to make vertex 0 become an element of the next block, the index of the block of vertices with degree 1, **b**[1], is decremented by one to include vertex 0 as its first element. We have **b**[1] = 1, and vertex 0 becomes the first element of the block of vertices with degree 1.

Table 2 shows the current status of arrays after swapping vertex 0 with 6. The updated values are shown in red color. As can be seen, **valid**[0] = $true$ because now the η-degree of vertex 0 is available. Now, vertex 6 is the first vertex in **A** and the algorithm processes it. The summary of the results is shown in Table 3. Since **valid**[6] = $true$ we can safely remove vertex 6. The corresponding index in array **gone** is set to true, and the coreness of vertex 6 is set to **d**[6], which is 1.

When a vertex v is removed, we process those neighbors u of v with a higher degree (lower-bound or exact) than v's (see lines 14-16), and decrement their degrees by one. Therefore, these neighbors should be moved one block to the left in **A**. This is done in constant time using the swap_left function (line 15) which is shown in Algorithm 2. For instance, vertex 3 is a neighbor of vertex 6 with degree 2, which is decremented by one, from 2 to 1, when vertex 6 is removed. Therefore, vertex 3 should be swapped to the block in the left in **A**, which contains vertices of degree 1. The process is similar to swapping to the right. However, when swapping to the left, vertex u is swapped with the first vertex, w in the same block in **A**. In addition, the positions of u and w are swapped in **p**. Then, the block index in **b** is incremented by one (line 7, Algorithm 2), making u the last element of the previous block.

Correctness of the algorithm. For every $v \in V$, and $C \subseteq V$, a vertex property function [3] is a function $\phi(v, C) : V \times 2^V \rightarrow \mathbb{R}$, and it is monotonic if $\forall C_1, C_2 \subseteq V : C_1 \subseteq C_2$ implies that $\phi(v, C_1) \leq \phi(v, C_2)$. According to the result by Batagelj and Zaversnik [3], for a monotonic vertex property function $\phi(v, C)$, the algorithm that repeatedly removes the vertex with smallest ϕ value gives the core decomposition. Since η-degree of a vertex is a monotonic vertex property function [6], then our peeling algorithm, which removes the vertex with smallest η-degree at each iteration of the algorithm, computes the desired core decomposition. Also, it should be noted that, the algorithm scans the next vertex in array **A** only when the η-degree of the previous vertex has been set. In addition, the lower-bounds never surpass the value of η-degrees. Thus, we conclude that the PA algorithm computes the desired core decomposition in probabilistic graphs.

Running time of the PA algorithm. The compute_swap function is dominated by two parts: **(1)** computing η-degrees takes $O(\eta\text{-deg}(u))$ for each vertex u **(2)** swapping a vertex to the proper block; each swap is done in constant time, and the maximum number of swaps required for a vertex u is $O(\eta\text{-deg}(u))$. However, as reported above, initially the difference between the lower-bounds obtained by Lyapunov CLT and the η-degrees is no more than one. Hence, either one or no swap is required initially. Thus, the compute_swap function takes in total

$$O\left(\sum_{v\in V}\sum_{u:(u,v)\in\mathcal{N}_v}\eta\text{-deg}(u)\right) = O\left(\sum_{v\in V} d_v\delta'\right) = O(m\delta'),$$

where m is the number of edges, and δ' is the maximum η-degree over all vertices at the time of their removal. The swap_left function swaps each vertex one block to the left in **A** in constant time. Therefore, the main cycle (Algorithm 1, line 6-19) takes $O(m\delta')$. In conclusion, the running time of the PA algorithm is $O(m\delta')$.

Index	0	1	2	3	4	5	6
d	0	1	1	2	1	1	0
A	0	6	1	2	4	5	3
p	1	3	4	7	5	6	2
b	0	2	6				
valid	false	false	false	false	false	false	false
gone	false	false	false	false	false	false	false

Table 1: Arrays d, b, A, p, valid, and gone in the PA algorithm for the graph in Figure 1.

Index	0	1	2	3	4	5	6
d	1	1	1	2	1	1	0
A	6	0	1	2	4	5	3
p	2	3	4	7	5	6	1
b	0	1	6				
valid	true	false	false	false	false	false	false
gone	false	false	false	false	false	false	false

Table 2: First step of the PA algorithm (after swapping vertex 0) for the graph in Figure 1.

Index	0	1	2	3	4	5	6
d	1	1	1	2	1	1	1
A	6	0	1	2	4	5	3
p	2	3	4	7	5	6	1
b	0	0	6				
valid	true	false	false	false	false	false	true
gone	false	false	false	false	false	false	false

Table 3: Second step of the PA algorithm executed on the graph in Figure 1.

5 SEQUENTIAL ALGORITHM (SA)

In this section we present a sequential algorithm (SA) which processes the vertices one-by-one, and as such, does not require the graph to be fully loaded into memory but rather one vertex at a time. Furthermore, as shown in Section 6, after only a fraction of iterations, SA is able to produce high quality results.

SA maintains bookkeeping information for each vertex and has a memory footprint of $O(n)$ as opposed to $O(m)$ for PA. More

Algorithm 1 PA k-core computation function

```
1: function K_CoreCompute(Graph G, η)
2:     initialize(d, b, p, A, G)
3:     gone ← False                                  ▷ all-false vector
4:     valid ← False                                 ▷ all-false vector
5:     i ← 1
6:     while i < n do
7:         v ← A[i]
8:         if valid[v] = true then
9:             gone[v] ← true
10:            for all u : (u, v) ∈ N_v do
11:                if d[u] = d[v] then
12:                    if valid[u] = false then
13:                        compute_swap(d, p, b, A, u, valid)
14:                if d[u] > d[v] then
15:                    swap_left(d, p, b, A, u)
16:                    valid[u] ← false
17:            i + +
18:        else
19:            compute_swap(d, p, b, A, v, valid)
20:    return d
```

Algorithm 2 PA swap to left function

```
1: function swap_left(d, p, b, A, u)
2:     du ← d[u], pu ← p[u]
3:     pw ← b[du] , w ← A[pw]
4:     if u ≠ w then
5:         A[pu] ← w, A[pw] ← u
6:         p[u] ← pw, p[w] ← pu
7:     b[du] + +, d[u] − −
```

Algorithm 3 PA swap to right function

```
1: function swap_right(d, p, b, A, u)
2:     du ← d[u], pu ← p[u]
3:     pw ← b[du + 1] − 1 , w ← A[pw]
4:     if u ≠ w then
5:         A[pu] ← w, A[pw] ← u
6:         p[u] ← pw, p[w] ← pu
7:     b[du + 1] − −, d[u] + +
```

Algorithm 4 η-degree computation and swap function

```
1: function compute_swap(d, p, b, A, u, valid)
2:     η_deg ← compute η-deg(u)
3:     valid[u] ← true
4:     diff ← η_deg − d[u]
5:     for j ← 1 to diff do
6:         swap_right(d, p, b, A, u)
```

specifically, SA adopts the "semi-external" model of computation, which assumes that for each vertex we can fit a small constant amount of information in main memory while the edges of the graph are stored on disk. As other works have shown, this model is practical for a large number of real-world, web-scale graphs, and widely adopted to handle other graph problems [32, 45, 50, 51].

Algorithms that are sequential and semi-external do exist for deterministic core-decomposition (see [24, 35, 45]). They are based on the idea of maintaining an upper-bound on each vertex's coreness. This upper-bound is initialized to the degree of each vertex, and after each iteration of the algorithm it is tightened further using a simple *locality property* until it reaches the exact core value. The locality property is as follows. The coreness of a vertex v can at most be the largest value k such that v has at least k neighbors with a value greater or equal to k. Locality-based tightening (LBT) lowers the bound of a vertex to be the number k described above.

Unfortunately, this idea does not work for probabilistic core decomposition. LBT does not necessarily converge to true core values of vertices. For an example, consider Figure 1, $\eta = 0.5$, and Table 4. We initialize the upper-bounds to the η-degree of each vertex. Then, we execute a round of LBT. The bound for vertex 3 is tightened to 2. This is because the largest k for which vertex 3 has at least k neighbors with a value at least k is 2. These neighbors are 1, 2, 4, and 5 with a bound equal to 2. Running one more iteration of LBT does not produce any further tightening. However, the true core number of vertex 3 is 1 not 2 (see Table 4, last column). In the following, we tackle the problem with a new procedure we call *probabilistic bound tightening* (PBT). PBT takes into consideration the edge probability values and uses an optimized version of dynamic programming to gradually tighten the upper bounds of vertices to the true coreness. While PBT by itself can be used to compute coreness, we combine PBT with LBT in order to speed up the convergence, since LBT is faster than PBT.

First we show that the obtained values at the end of LBT are always an upper-bound to the true coreness values.

Proposition 1. *Let $\mathcal{G} = (V, E, p)$ be a probabilistic graph. Also, for each vertex $v \in V$, let k_v be the true coreness of v, and $\bar{k}_v$ be the value assigned to v at the end of the LBT phase. Then, $\bar{k}_v \geq k_v$.*

Proof. Once a LBT phase terminates, and the coreness of v is fixed to $\bar{k}_v$; then there should be a subset $V_{\bar{k}_v}$ of neighbors of v with the size at least $\bar{k}_v$, and $\forall u \in V_{\bar{k}_v} : \bar{k}(u) \geq \bar{k}_v$, where $\bar{k}(u)$ is the coreness assigned to u by LBT. However, considering the existence probability of edges incident to v, the value for $\Pr[d_{\mathcal{G}(V_{\bar{k}_v})}(v) \geq \bar{k}_v]$ can be less than η, where $\mathcal{G}(V_{\bar{k}_v})$ is the induced subgraph by $V_{\bar{k}_v}$. On the other hand, since $\Pr[d(v) \geq k]$ is monotonically non-increasing with the value of k, the true coreness of v should be in the interval $[0, \bar{k}_v]$ such that $\Pr[d_{\mathcal{G}(V_{\bar{k}_v})}(v) \geq k] \geq \eta > \Pr[d_{\mathcal{G}(V_{\bar{k}_v})}(v) \geq \bar{k}_v]$. Thus, $\bar{k}_v \geq k_v$. □

Once a LBT phase terminates, PBT starts to check whether the obtained value for each vertex is the true coreness of the vertex or not. If not, the gap is tightened further. We run LBT and PBT repeatedly, one after the other, until the core value for each vertex reaches a fixed point.

Before formally giving the algorithm, we present an example to illustrate how the SA algorithm works. We consider the probabilistic graph in Figure 1 and $\eta = 0.5$. LBT for this example was discussed earlier and the vertex values at the end of it are given in Table 4, third column. As can be seen, there are some vertices whose bounds are different from their true coreness (e.g. vertices 1 and 3). Therefore, the SA algorithm starts PBT to close this gap by checking for exactness of each bound to the true coreness of the vertex. For instance, consider vertex 1. We should check whether the coreness of it can be equal to 2 or not. In probabilistic core decomposition, a vertex v has coreness k if

$\Pr[d_{\mathcal{H}}(v) \geq k] \geq \eta$, where $\mathcal{H}$ is a maximal subgraph in which each vertex has degree of at least k. Checking $\Pr[d_{\mathcal{H}}(1) \geq 2]$, we see that this probability does not pass threshold $\eta = 0.5$, and therefore vertex 1 cannot have coreness of 2 in subgraph $\mathcal{H}$ which contains the neighbors 2 and 3. We perform the check of $\Pr[d_{\mathcal{H}}(v) \geq k]$ using an optimized version of dynamic programming explained later. Vertex 0 cannot belong to the subgraph because its bound is less than 2. For vertex 1, the maximum value, for which the probability passes the threshold, is equal to 1. As a result, the bound on the coreness of vertex 1 is updated to 1. The same process is done for vertex 3. The values obtained at the end of PBT are shown in the fourth column of Table 4, which contains exactly the true core value of each vertex.

v	η-degree	LBT-Round 1	PBT-Round 1	Coreness
0	1	1	1	1
1	2	2	1	1
2	2	2	2	2
3	3	2	1	1
4	2	2	2	2
5	2	2	2	2
6	1	1	1	1

Table 4: Upper-bounds obtained at LBT and PBT phase of SA. $\eta = 0.5$ ***for the example. LBT and PBT correspond to locality-based and probability bound tightening, respectively.***

SA algorithm description. The main cycle of SA is given in Algorithm 5. We define two Boolean variables, *PBT_phase* and *etadegree_change*. Variable *PBT_phase* is used to determine whether a PBT phase has been started or not. The variable *etadegree_change* is used to record whether there is some vertex (in PBT phase) with its η-degree changed or not. Initially both of them are set to *false*. Then, we invoke LBT. Once this process terminates (line 4), the vertex value (upper-bound) of all the vertices should be checked for the possibility of gap between the current vertex value and its true coreness. In fact, for each vertex v it should be investigated whether the degree of v can be greater than or equal to its current vertex value with probability no less than η or not. If so, the current vertex value should be the same as its η-deg(v). Checking whether a vertex should be processed (in PBT) or not is done using the Boolean array *check* which contains a flag for each vertex. If the flag is set to *true*, its vertex value needs to be checked, using the *PBT* function (see line 10). Then, the algorithm enters a PBT phase (lines 8-10). We make a clone, *checkNow*, of *check*; then we reinitialize *check* to the all-false vector (lines 8-9). The clone is needed to be used in the *for* loop in the *PBT* function (line 3, Algorithm 6). If after a PBT phase terminates there is some vertex with its η-degree not equal to its current vertex value, a new LBT phase is started again (line 15) and this process continues until a fixed point is obtained for all the vertices (lines 11-12).

PBT is given in Algorithm 6. The implementation iterates over each vertex v, and checks if there is a gap between v's value and its true coreness; if so, that vertex should be processed. In this process, only the neighbors of v whose current values are greater or equal to v's value are considered because it is only them that can contribute to v's coreness. We compute $\Pr[d(v) \geq C[v]]$ and check whether it passes threshold η or not. If $\Pr[d(v) \geq C[v]] < \eta$, then, since we know that $\Pr[d(v) \geq k]$ is non-increasing with k, the maximum k for which the probability passes the threshold is returned and stored in variable *localValue* (line 5). In this case, all the values from $(C[v] - 1)$ down to *localValue* should be checked as candidate values for the coreness of v and the maximum value (variable *max* in line 16), is chosen as the new value of v. This way we make sure that the algorithm does not go below the real coreness value at each PBT step (lines 8-15). For each value j in the *for* loop (line 8) we check whether $\Pr[d(v) \geq j] \geq \eta$ (line 9). If so, *max* is set to j (line 10) and the vertex value is updated to *max* (line 16). Otherwise, the value for which the probability passes the threshold is stored in variable *value* (line 13), and if it is greater than *max*, the latter is updated to the former (lines 14-15).

It should be noted that in the *for* loop, similar to what we do in lines 4-5 of Algorithm 6, we should compute $\Pr[d(v) \geq j]$ and check if it passes the threshold or not. For this, we use an optimized dynamic programming process as follows. Variable j starts from $C[v] - 1$ and goes down to *localValue*. In order to avoid computing these probabilities from scratch, once we compute $\Pr[d_{\mathcal{H}}(v) \geq C[v]]$, we cache the following probabilities $\Pr[d_{\mathcal{H}}(v) = 0], \cdots, \Pr[d_{\mathcal{H}}(v) = C[v]]$, where $\mathcal{H}$ contains all the neighbors of v whose upper-bound is at least $C[v]$. Then, since for $j = C[v] - 1$, $\mathcal{H}'$ contains all the neighbors whose upper-bound is exactly equal to j (we denote this set by V_j) and higher, we can have $\mathcal{H}' = V_j \cup \mathcal{H}$. Therefore, to compute $\Pr[d_{\mathcal{H}'}(v) \geq j]$ we use the computed probabilities in $\mathcal{H}$ and only consider vertices in V_j. This way we optimize DP to compute the probabilities very fast since V_j is typically small (e.g., about 100 in our later evaluated large-scale graphs). For the next iteration, we store all the probabilities computed in the previous iteration and use them to compute new probabilities. In the following we describe our DP process in more detail.

Let assume that we have computed $\Pr[d_{\mathcal{H}}(v) = k]$, where $k = 0, 1, ..., j$, and $\mathcal{H}$ is a subgraph of the input probabilistic graph in which each vertex has core value at least j. Also, let V_{j-1} contain all the vertices (including the neighbours of v) whose core value is exactly $j-1$. Thus, $\mathcal{H}' = \mathcal{H} \cup V_{j-1}$ is the subgraph whose vertices have core value at least $j-1$. We denote $\{e_1, ..., e_x\}$ to be the set of edges incident to v such that for each $e_i = (u_i, v)$, $u_i \in V_{j-1}$, where $i = 1, ..., x$. In order to evaluate $\Pr[d_{\mathcal{H}'}(v) = k']$, where $0 \leq k' \leq k$, and avoid from scratch computation, we denote the degree of v in the subgraph $\mathcal{H}'$ by $d(v \mid (\mathcal{H} \cup \{e_1, ..., e_x\}))$. Then, it holds that:

$$\begin{aligned} &\Pr[d(v \mid (\mathcal{H} \cup \{e_1, ..., e_x\})) = k'] = \\ &= p_{e_x} \Pr[d(v \mid (\mathcal{H} \cup \{e_1, ..., e_{x-1}\})) = k' - 1] + \\ &+ (1 - p_{e_x}) \Pr[d(v \mid (\mathcal{H} \cup \{e_1, ..., e_{x-1}\})) = k']. \end{aligned} \tag{16}$$

Letting $T(x, k') = \Pr[d(v \mid (\mathcal{H} \cup \{e_1, ..., e_x\})) = k']$, we have the following recursive formula:

$$T(x, k') = p_{e_x} T(x-1, k'-1) + (1 - p_{e_x}) T(x-1, k') \tag{17}$$

with the following base cases:

$$\begin{cases} T(0, k') = \Pr[d_{\mathcal{H}}(v) = k'], & 0 \leq k' \leq k \\ T(x, -1) = 0, \end{cases} \tag{18}$$

As can be seen, in the base case of the recursive formula, we are using the previously computed probabilities to compute the probabilistic degree of vertex v in the new subgraph, which results in saving time significantly.

Since the vertex value of v is updated, all its neighbors with value at least the vertex value of v should be checked for validity of their vertex value. We show in the following that the PBT estimate of the coreness eventually equals the true coreness for each vertex.

Algorithm 5 SA k-core computation function

```
1:  function SA_CORE_COMPUTATION(Graph 𝒢)
2:      PBT_phase ← false
3:      C ← 0                                   ▷ array of core values
4:      LBT()
5:      check ← True                            ▷ all-true vector
6:      etadegree_change ← false
7:      while true do
8:          checkNow ← check.clone()
9:          check ← False                       ▷ all-false vector
10:         PBT(𝒢, checkNow)
11:         if etadegree_change = false then
12:             break
13:         else
14:             PBT_phase = true
15:             LBT()
16:             etadegree_change ← false
17:     return cores
```

Algorithm 6 SA probabilistic bound tightening function

```
1:  function PBT(𝒢, checkNow)
2:      for all v ∈ V do
3:          if checkNow[v] = true then
4:              if Pr[d_ℋ(v) ≥ C[v]] < η then
5:                  localValue ← compute η-deg_ℋ(v)
6:                  max ← localValue
7:                  m ← C[v] − 1
8:                  for all j ← m down to localValue do
9:                      if Pr[d_ℋ'(v) ≥ j] ≥ η then
10:                         max ← j
11:                         break
12:                     else
13:                         value ← compute η-deg_ℋ'(v)
14:                         if value ≥ max then
15:                             max ← value
16:                 C[v] ← max
17:                 etadegree_change ← true
18:                 for all u : (u, v) ∈ 𝒩_v do
19:                     if C[u] ≥ C[v] then
20:                         check[u] ← true
```

Correctness of the SA algorithm. We prove this by contradiction. Suppose that after the last PBT phase, there is vertex u_1 such that $k(u_1) = k_1$ (real coreness) and $core[u_1] = k' > k_1$, where $core$ is the coreness assigned to u_1 after PBT phase. Since $k(u_1) = k_1$, k_1 is the maximum value such that $\Pr[d_{\mathcal{H}}(u_1) \geq k_1] \geq \eta$, and $\Pr[d_{\mathcal{H}'}(u_1) \geq i] < \eta$ for all $i > k_1$, where $\mathcal{H}$ and $\mathcal{H}'$ are the maximal induced subgraphs in which each vertex has degree at least k_1 and i, respectively, with probability no less than η. Based on the properties of cores of a graph, we know that $\mathcal{H}' \subseteq \mathcal{H}$. If all the neighbors of u_1 in $\mathcal{H}$ have coreness k_1, then u_1 will not have any neighbors with coreness greater than k_1 in $\mathcal{H}'$, so u_1 eventually sets $core[u_1]$ equal to k_1, which is a contradiction. The similar argument holds if all the neighbors of u_1 in $\mathcal{H}$ have coreness greater than k_1. Since $k(u_1) = k_1$ and $core[u_1] = k' > k_1$, there should exist a neighbor u_2 with $k(u_2) = k_1$ and $core[u_2] > k_1$ such that $\Pr[deg_{\mathcal{H}'\cup\{u_2\}}(u_1) \geq k'] \geq \eta$, because otherwise $\Pr[d_{\mathcal{H}'}(u_1) \geq k'] < \eta$. In fact, the existence of this neighbor will contribute to the assigned coreness of u_1 to be greater than k_1.

Now by reasoning similar to [35], we can build a sequence of vertices $S = \{u_i, u_{i+1}, u_{i+2}, ..., u_j = u_i\}$ connected to each other with $k(u_i) = k_1$ and $core[u_i] > k_1$. For each vertex u_i in S, let V_i be the set of all neighbors of u_i in $\mathcal{H}'$. Now, we can define a set $U = S \cup \bigcup_{u_i \in S} V_i$. The corresponding induced subgraph $\mathcal{G}(U)$ is a k'-core, because all the vertices in V_i have coreness at least k' with probability greater than or equal to η. Also, since for each vertex u_i in S, $\Pr[d_{V_i\cup\{u_{i+1}\}}(u_i) \geq k'] \geq \eta$, we have that $\mathcal{G}(U)$ is a k'-core where $k' > k_1$. Hence, we find a subgraph whose vertices have coreness $k' > k_1$ which is a contradiction because we assumed that each vertex in S has coreness k_1. Therefore, k_1 is not a maximal (i.e. true) coreness.

Running time of the SA algorithm. The time complexity of the algorithm is dominated by time complexity of the PBT because LBT has a time complexity of $O(N - K + 1)$ [35], where K is the number of vertices with minimal degree (in probabilistic graphs it refers to η-degree). To analyze the time complexity of a PBT step (Algorithm 6), let Δ be the maximum upper-bound on the η-degree over all the vertices in all the probabilistic bound tightening steps. Lines 4 and 5 (η-degree computation) are done simultaneously, and take time $O(d_v C[v])$ for each vertex v, where $C[v]$ is the upper-bound on the η-degree of v. In practice this computation is fast because the η-degree computation is performed on the subgraph $\mathcal{H}$ which contains only those neighbors of v whose upper-bound is at least $C[v]$ (not all the d_v vertices). In the worst case the inner loop is repeated $C[v]$ times, and each time the η-degree computation and the checking of the probability threshold (lines 9 and 13) are performed similarly to what explained above. Therefore, the time complexity of this part is: $\sum_{j=1}^{C[v]} O(jd_v)$. It should be noted that at each iteration in the for loop, we use the previously computed probabilities to avoid computing η-degrees from scratch. However, here we consider the worst case analysis of the algorithm. The time complexity of lines 18-20 is $O(d_v)$. As a result, each PBT iteration takes $\sum_{v\in V}\left(O(C[v]d_v) + \sum_{j=1}^{C[v]} O(jd_v) + O(d_v)\right) = \sum_{v\in V}\left(O(\Delta d_v) + O(\theta d_v) + O(d_v)\right)$, where $\theta = \Delta^2$. Therefore, the time complexity of each PBT round is $O(m\theta)$, where m is the total number of edges. In the worst case, we assume that at each PBT round, the difference between the actual coreness of a vertex and its initial estimate (the initial η-degree) decreases by one unit. Thus, in the worst case $\Gamma = \sum_{v\in V} \eta\text{-deg}(v)$ PBT steps are required. Therefore, the total running time can be expressed as: $O(\Lambda m)$, where $\Lambda = \Gamma\theta$.

As in [35], the complexity upper bounds for such iterative algorithms are not representative of practical performance. In practice, LBT and PBT are fast, and the number of iterations is only a handful, thus SA is an efficient algorithm for large datasets while requiring only $O(n)$ memory footprint.

6 EXPERIMENTS

In this section, we present our experimental results. Our implementations are in Java and the experiments are conducted on a commodity machine with Intel i7, 2.2Ghz CPU, and 12Gb RAM, running Ubuntu 14.03. The hard disk is Seagate Barracuda ST31000524AS 2TB 7200 RPM.

The statistics for all of the datasets we consider are shown in Table 5. We obtained flicker, dblp, and biomine from the authors of [6], and the rest of the datasets from Laboratory of Web Algorithmics.[1] The datasets are divided by horizontal lines according to their size, small (S), medium (M), large (L), and extra large (XL).

[1] http://law.di.unimi.it/datasets.php

We use the Webgraph framework [5] to store these datasets. The flickr, dblp, and biomine datasets already contained probability values. For the other datasets we generated probability values uniformly distributed in $[0, 1]$.

We evaluate our algorithms in three important aspects. First is numerical stability. As [6] points out, using probability values may lead to numerical instability. We discuss this in Subsection 6.1. Second is the quality of our η-degree lower-bounds using Lyapunov CLT. We evaluate and discuss this in Subsection 6.2. Third is the efficiency of our proposed algorithms. We show our performance results in Subsection 6.3.

6.1 Numerical Stability

For evaluating numerical stability we refer to the results of η-degree computations shown in Table 6. The table contains results for $\eta = 0$, $\eta = 10^{-9}$ and $d_v = 100$, $d_v = 1000$. For each combination of η and d_v we compute η-degrees for 1000 randomly selected vertices. We show results for twitter-2010, but any of the other datasets above can be used for this experiment to obtain similar results.

Using the BigDecimal class in Java we adjust the numerical precision to different levels when using DP for computing η-degrees. We compare numerical results of DP using different precision levels in Java, where DPU (DP with unlimited precision) is the highest level of precision and thus the gold standard. The DP128 and DP256 columns in the table correspond to computing η-degrees using DP by setting precision to 128 bit and 256 bit, respectively. We observe that the more we increase the level of precision, the longer it takes to perform the computation. For example, when executing the DP algorithm for computing η-degrees for vertices with $d_v = 1000$, it takes more than 7000 times longer to perform the computation using unlimited precision than using the default (plain) number computation in Java.

Following [6], we start by setting $\eta = 0$ (top two parts). For this η the (k, η)-core decomposition of a probabilistic graph should coincide with the core decomposition of the deterministic graph derived by ignoring probabilities. The accuracy can thus be measured by comparing, for each vertex, the $(k, 0)$-core number with the core number obtained on the deterministic version of the graph.

We can see that for $\eta = 0$, we need a lot of precision (bits) to achieve error free computation. For example, when $d_v = 100$, we need at least 256-bit precision, whereas when the degree is 1000, we need DP with unlimited precision.

The DPLog2 column of the table shows the results if we operate in logarithmic space[2]. We can see that DP operating in logarithmic space is very stable and never produces any error at all, while being much faster than the DP variants operating with specified precision.

We test $\eta = 0$ to compare our results to previous work [6]. However, the situation changes significantly if η is greater than zero even by a very small amount. If we set $\eta = 10^{-9}$ (one billionth) or higher, we never get an error even for plain DP which is much faster than any other DP variant including the one in log space (see the two bottom parts). We tested with η in $[10^{-9}, 0.5]$ and obtained similar results.

The last column of Table 6 shows the accuracy of η-degree computation when using Lyapunov CLT instead of DP. For $\eta = 0$, computing η-degrees using CLT is error-free. More interesting are the bottom two parts for η greater than zero. For vertices with $d_v = 100$, CLT makes errors in computation (on 43% of the vertices), however, those errors are small; only 1% on average of the η-degree value.

When considering vertices with $d_v = 1000$, the computation using CLT is error-free and furthermore it is orders of magnitude faster than the variants of DP. Again, we also tested with a variety of η levels and obtained similar results. Guided by the above, in our algorithms, we set the threshold on d_v to start using Lyapunov CLT for computing η-degrees at 1500. Recall that the η-degree computation is needed in PA when a vertex becomes candidate for removal, while in SA it is needed when initializing vertex values.

In summary, regarding numerical stability, our contribution is to show that DP is sensitive to the setting of η value and becomes error-free once η is greater than zero even by a small amount. This was not investigated thoroughly before. On the other hand, CLT is resilient to different η values (even $\eta = 0$). With respect to efficiency in computing η-degrees, when $d_v \geq 1000$, CLT can be used to produce error-free computations orders of magnitude faster than all the DP variants.

Name	$\|V\|$	$\|E\|$
flickr	24,125	300,836
dblp	684,911	2,284,991
cnr-2000	325,557	2,738,969
biomine	1,008,201	6,722,503
ljournal-2008	5,363,260	49,514,271
arabic-2005	22,744,080	553,903,073
uk-2005	39,459,925	783,027,125
twitter-2010	41,652,230	1,202,513,046

Table 5: Dataset Statistics

	DP	DP128	DP256	DPU	DPlog2	CLT
NE	100%	99%	0%	0%	0%	0%
AE	8%	5%	0%	0%	0%	0%
AT	0.09	11.44	12.57	27.71	1.56	0.05
NE	100%	100%	100%	0%	0%	0%
AE	40%	35%	28%	0%	0%	0%
AT	3.3	1238	1499	26355	222	0.1
NE	0%	0%	0%	0%	0%	43%
AE	0%	0%	0%	0%	0%	1%
AT	0.19	14.41	15.34	43.86	2.14	0.27
NE	0%	0%	0%	0%	0%	0%
AE	0%	0%	0%	0%	0%	0%
AT	3.7	1244	1382	23934	171	0.3

Table 6: Error statistics and average running time for different precision levels for $\eta = 0$ and $\eta = 10^{-9}$. NE stands for Number of Errors, AE for Average Relative Error, and AT for Average Time (ms). Specifically, AE=$\|error\|/true\_value$.
1st part: $\eta = 0$, $d_v = 100$; **2nd part:** $\eta = 0$, $d_v = 1000$; **3rd part:** $\eta = 10^{-9}$, $d_v = 100$; **4th part:** $\eta = 10^{-9}$, $d_v = 1000$
DP: *DP using plain numbers in Java;* **DP128, DP256, DPU:** *DP using BigDecimal in Java and setting the precision to 128 bits, 256 bits, and unlimited, respectively;* **DPlog2:** *DP doing computations in log space.*

6.2 Accuracy of CLT as a lower-bound

Here we investigate the quality of the lower-bounds on η-degrees obtained using CLT. Namely, we compare CLT with another method for deriving lower-bounds used in [6], which utilizes a

[2] https://en.wikipedia.org/wiki/Logarithm

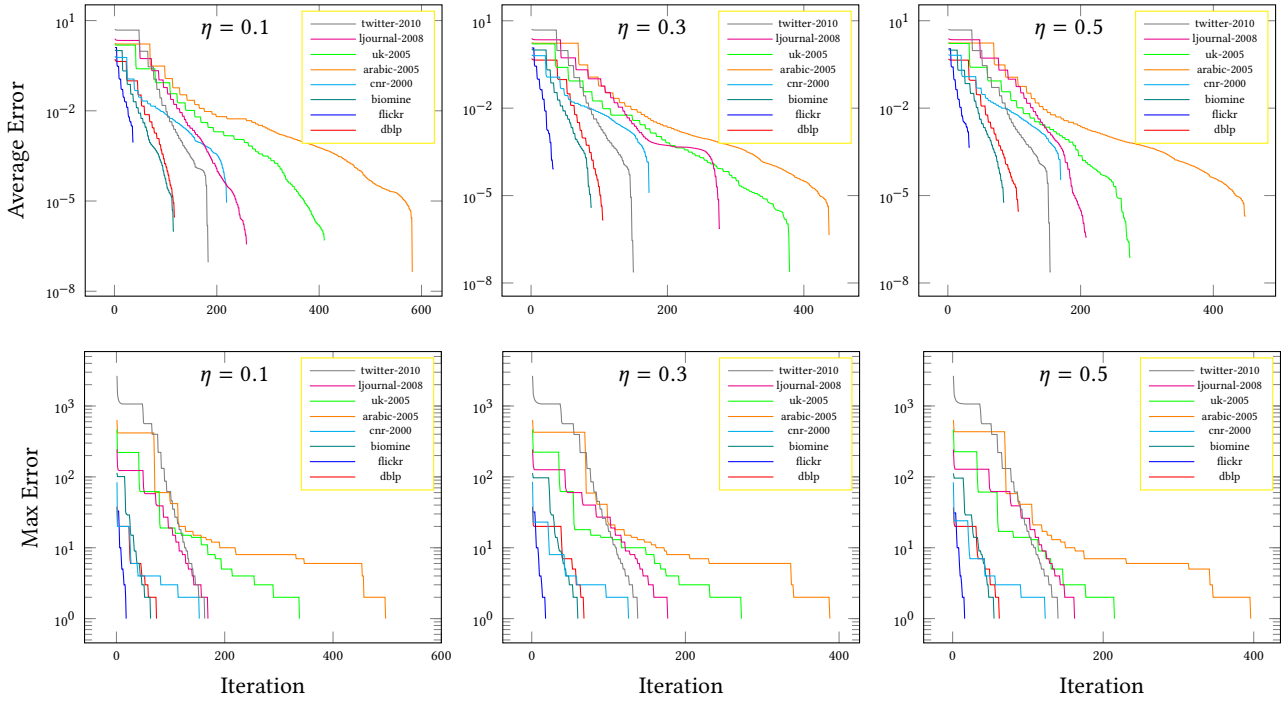

Figure 2: Average error (average of difference from true core coreness) and max error (maximum difference from true coreness) versus iterations for different values of η.

Dataset	Max Error	
	CLT	Beta Function
biomine	1	1,663
cnr-2000	1	9,056
ljournal-2008	1	9,453

Table 7: Maximum error of CLT and regularized beta function for selected datasets.

formula based on the regularized beta function.[3] We computed initial η-degrees for biomine, cnr-2000, and ljournal-2008 with $\eta = 0.1$ using DP, Lyapunov CLT, and regularized beta function. Then, we computed the maximum error for CLT and regularized beta function. We report the results in Table 7. As can be seen, the maximum error for CLT for all the datasets is one which means that the difference between the values obtained by CLT and the true values is either zero or at most one. On the other hand the max error for the regularized beta function is big, in the order of thousands.

The small value of error for CLT is of great importance in the main cycle of the PA algorithm where each vertex is processed based on its lower-bound before its true η-degree is computed. To summarize, since the difference between each vertex's lower-bound (computed by CLT) and its true η-degree is small, we only need to do a small number of swaps to place a vertex in the proper place in the array $\mathbf{A}$ resulting in significant savings in running time.

6.3 Efficiency of the Proposed Algorithms

Table 8 shows the running times of the PA (left) and SA (right) algorithms on the selected datasets. For twitter-2010 we report the results for different values of η ranging from 0.1 to 0.5. For the other datasets, we only show results for $\eta = 0.1$ and omit results for $\eta = 0.2, \ldots, 0.5$, since they are similar to those for $\eta = 0.1$ and their nature is the similar to what we see for twitter-2010.

For the small and medium datasets, PA produced the results in about 2 sec for the small and 4 to 6 sec for the medium datasets. PA also performed well on the large datasets; biomine and ljournal-2008, computing the core decomposition of these two graphs on average in only 40 sec and 2 min, respectively. Notably, for biomine for instance, our algorithm is 32 times faster than the algorithm of [6] (see also Table 10 which we discuss later).

The running time of PA is good on the very large datasets as well. On uk-2005 and arabic-2005, PA completed in about 28 min and 42 min, respectively; less than one hour; in contrast the state-of-art [6] was not able to complete for these datasets in our machine after one day. For twitter-2010, which is much larger than uk-2005 and arabic-2005, with a maximum η-degree of 1, 500, 282, our PA algorithm completed in around two hours, which is impressive for processing such a big dataset on our consumer-grade machine.

The running times of SA are shown in Table 8 (right part). For the small and medium datasets the total running times of SA were similar to those of PA. For the large datasets, SA needed more time, 2-3 times more, than PA. However, we recall that SA has a much smaller memory footprint than PA, namely $O(n)$ as opposed to $O(m)$ for PA. As such we are trading time for space when using SA over PA, a beneficial feature to have when using low-cost, cloud-server machines.

Another benefit of using SA is that it can produce good approximate results after only a small fraction of total iterations. We discuss this more later in this section.

Effect of η values. We observe that the total running time does not change much as the value of η increases. This is because when η increases, the η-degree of a vertex might not change or slightly decrease and as such this does not have a significant effect on the total running time of the algorithms. This is also evident in

[3] http://mathworld.wolfram.com/RegularizedBetaFunction.html

Table 9 where the average coreness (kavg) and the maximum coreness (kmax) decrease only slightly as η increases. As before, we report the average coreness, the maximum coreness, and the maximum η-degree for twitter-2010 for $\eta = 0.1, \ldots, 0.5$, whereas for the other datasets we only show the values for $\eta = 0.1$.

Comparing with the algorithm of [6]. We also compared the running times of our algorithms, PA and SA, to that of the algorithm of [6] (which we call BGKV[4]), for $\eta = 0.1$. We considered the (k, η)-core implementation made available to us by the authors of [6] with numbers represented by using BigDecimal in Java. From this, we also created a second version, which we call BGKV-2, where we replaced the computations using BigDecimal with plain computations in Java. This is because as shown in Subsection 6.1, for η greater than 0, the use of BigDecimal for DP is not warranted.

Table 10 summarizes the running times on three datasets, flickr, dblp and biomine. For biomine, which is a large dataset, we were only able to use up to 64 precision bits. We can see that both PA and SA are significantly faster than BGKV. For example, for biomine our algorithms are more than 32 times faster than BGKV.

BGKV-2 is faster than BGKV. For the small datasets, flickr and dblp, it is even faster than PA and SA. This can be attributed to the fact that since the memory footprint is small, the benefit of using our algorithms is outweighed by the overhead of using the Webgraph compression structures in PA and SA.

The situation changes significantly as the size of the datasets grows. For biomine, BGKV-2 is about twice slower than PA and SA. For the rest of the datasets that are bigger than biomine, BGKV-2 cannot run to completion in our machine after one day. For those datasets, our algorithms, PA and SA, are the only algorithms that can produce results in a matter of minutes or few hours (for twitter-2010), Table 8.

Convergence speed of SA. To further investigate the execution of SA as it unfolds with time, we look at average error and maximum difference (max error) from the true core value over the sequence of iterations (see Figure 2). As shown by the plots in the top part of the figure, the average error sharply decreases for all the datasets which we consider, except for uk-2005 and arabic-2005 whose average errors decrease more gradually.

Similar to the average error, the maximum difference from the true core value (shown in the plots in the bottom part) drops quickly, becoming 1 in only a fraction of the total number of iterations. Furthermore, as the value of η increases, the total number of iterations required decreases.

These results show that SA produces approximate results of good quality in only a fraction of iterations needed for completion. For instance, for arabic-2005 with $\eta = 0.1$, the average error drops below 0.01 at iteration 200, only one third of the total number of required iterations (about 600, see the end of the curve). Depending on the application domain this can be a desirable property during data analysis.

7 RELATED WORK

Among different notions of cohesive subgraphs, k-core is one of the most popular (cf. [1, 24, 30, 44, 46]. Other definitions of dense subgraphs such as maximal cliques can also be computed using k-core decomposition [16]. In deterministic graphs, the computation of k-core has been well studied. Batagelj and Zaversnik [3] give an efficient peeling algorithm for deterministic core decomposition. Montresor et al. [35] give a distributed algorithm for deterministic core decomposition and introduce the concept of locality-based bound tightening. Wen et al. [45] propose I/O efficient core decomposition algorithms which only allow node information to be loaded in memory. Khaouid et al. [24] consider deterministic core decomposition of large networks on a single PC. Sariyuce et al. [39] propose incremental k-core decomposition algorithms for dynamic graph data, in which edges are added/deleted on a regular basis. In a similar setting, a distributed k-core decomposition and maintenance algorithms are proposed in [2]. Core decomposition in large temporal graphs is addressed in [47].

[4] Abbreviation using the first letters of the authors' names.

Dataset	η	PA-Running Time	SA-Running Time
flickr	0.1	2.05	1.88
dblp	0.1	5.81	9.57
cnr-2000	0.1	3.99	8.49
biomine	0.1	40	40
ljournal-2008	0.1	120	342
arabic-2005	0.1	2,539	2,513
uk-2005	0.1	1,709	1,961
twitter-2010	0.1	8,096	21,662
	0.2	8,547	20,268
	0.3	8,358	19,670
	0.4	8,364	20,544
	0.5	8,929	20,111

Table 8: Running times (sec) of PA and SA. Numbers less than 10 have been rounded to 2 decimal places, and those above 10 have been rounded to the nearest integer.

Dataset	**η-degmax**	**kmax**	**kavg**	**η**
flickr	78	46	3.70439	0.1
dblp	162	26	1.99825	0.1
cnr-2000	9,255	38	5.77113	0.1
biomine	6,269	79	3.08697	0.1
ljournal-2008	9,664	156	5.4735	0.1
arabic-2005	287,949	1,088	15.2903	0.1
uk-2005	888,658	274	13.0279	0.1
twitter-2010	1,500,282	986	15.7391	0.1
	1,499,970	976	15.0873	0.2
	1,499,744	970	14.6407	0.3
	1,499,552	964	14.2919	0.4
	1,499,372	959	13.9927	0.5

Table 9: Maximum η-degree, maximum probabilistic coreness, average probabilistic coreness, value of the threshold η.

Algorithm		flickr	dblp	biomine
BGKV	pr=64	18	90	2,493
	pr=128	27	133	N.P.
	pr=256	35	148	N.P.
BGKV-2		0.49	4.26	85
PA		2.05	5.81	40
SA		1.88	9.57	40

Table 10: Running time (sec) of the algorithm in [6] with BigDecimal (BGKV), and without (BGKV-2) versus PA and SA. "pr" is the precision (bits) used. BGKV cannot run for biomine to completion after one day (we use N.P. for "Not Possible"). BGKV-2 is faster for the small datasets, flickr and dblp, but twice slower for biomine than PA and SA. For the rest of the datasets that are bigger than biomine, both BGKV and BGKV-2 cannot run to completion in our machine after one day. Our algorithms PA and SA can produce results for every dataset, see Table 8.

For probabilistic graphs, the generalization of k-core is the notion of (k, η)-core introduced by Bonchi et al. [6], which we discussed in detail throughout the paper. Other notions of cohesive subgraphs are studied in probabilistic setting [18, 36, 55]. [36, 53] focus on the problem of finding k vertex sets with the largest maximal-clique probabilities. Truss decomposition as another notion of a cohesive subgraph has been studied in [18, 55].

Significant research has been done in mining and querying probabilistic graphs. Reachability is addressed in [12, 20, 22, 23]. Shortest paths are studied in [38, 48] and frequent subgraph mining in [10, 37, 52, 54]. Clustering analysis is investigated in [26, 31] and subgraph similarity in [49].

8 CONCLUSIONS

We presented two efficient algorithms, PA and SA, for computing the core decomposition of probabilistic graphs at web scale. An important contribution of this work is the use of Lyapunov Central Limit Theorem in these algorithms to compute tail probabilities for η-degrees. We evaluated our algorithms, and showed that they are efficient and numerically stable. Our algorithms were considerably faster than the-state-of-the-art for large datasets. For datasets larger than biomine, our algorithms PA and SA, were the only algorithms able to run to completion on a consumer grade machine. In particular, PA was able to compute probabilistic core decomposition for uk-2005, arabic-2005, and twitter-2010 in 28 min, 42 min and 2.2 hours, respectively, which is impressive for such large datasets. SA has smaller memory footprint and can produce approximate results of high quality in only a fraction of iterations needed for full completion.

REFERENCES

[1] L. Antiqueira, O. N. Oliveira Jr, L. da Fontoura Costa, and M. D. G. V. Nunes. 2009. A complex network approach to text summarization. *Inf. Sci.* 179, 5 (2009), 584–599.

[2] S. Aridhi, M. Brugnara, A. Montresor, and Y. Velegrakis. 2016. Distributed k-core decomposition and maintenance in large dynamic graphs. In *Proc. DEBS*. ACM, 161–168.

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

[4] P. Billingsley. 1995. *Mathematical Methods of Statistics* (3 ed.). Wiley.

[5] P. Boldi and S. Vigna. 2004. The webgraph framework I: compression techniques. In *Proc. WWW'04*. ACM, 595–602.

[6] F. Bonchi, F. Gullo, A. Kaltenbrunner, and Y. Volkovich. 2014. Core decomposition of uncertain graphs. In *Proc. SIGKDD*. ACM, 1316–1325.

[7] Ch. Borgs, M. Brautbar, J. Chayes, and B. Lucier. 2012. Influence Maximization in Social Networks: Towards an Optimal Algorithmic Solution. *CoRR* abs/1212.0884 (2012).

[8] C. Budak, D. Agrawal, and A. El Abbadi. 2011. Limiting the spread of misinformation in social networks. In *Proc. WWW'11*.

[9] G. Cavallaro. 2010. Genome-wide analysis of eukaryotic twin CX 9 C proteins. *Mol. Biosyst.* 6, 12 (2010), 2459–2470.

[10] Y. Chen, X. Zhao, X. Lin, and Y. Wang. 2015. Towards frequent subgraph mining on single large uncertain graphs. In *Proc. ICDM*. IEEE, 41–50.

[11] J. Cheng, Y. Ke, Sh. Chu, and MT. Özsu. 2011. Efficient core decomposition in massive networks. In *Proc. ICDE*. IEEE, 51–62.

[12] Y. Cheng, Y. Yuan, L. Chen, and G. Wang. 2015. The reachability query over distributed uncertain graphs. In *Proc. ICDCS*. IEEE, 786–787.

[13] H. Cramér. 1946. *Mathematical Methods of Statistics*. PUP.

[14] M. T. Dittrich, G. W. Klau, A. Rosenwald, T. Dandekar, and T. Müller. 2008. Identifying functional modules in protein–protein interaction networks: an integrated exact approach. *BOINFP* 24, 13 (2008), i223–i231.

[15] J. Dong and S. Horvath. 2007. Understanding network concepts in modules. *BMC system biology* 1, 1 (2007), 24.

[16] D. Eppstein, M. Löffler, and D. Strash. 2010. Listing all maximal cliques in sparse graphs in near-optimal time. In *ISAAC*. Springer, 403–414.

[17] A. Goyal, F. Bonchi, and L. V. Lakshmanan. 2010. Learning influence probabilities in social networks. In *Proc. WSDM*. ACM, 241–250.

[18] X. Huang, W. Lu, and L. V. Lakshmanan. 2016. Truss decomposition of probabilistic graphs: Semantics and algorithms. In *Proc. SIGMOD*. ACM, 77–90.

[19] R. Jin, L. Liu, and Ch. Aggarwal. 2011. Discovering highly reliable subgraphs in uncertain graphs. In *Proc. SIGKDD*. ACM, 992–1000.

[20] R. Jin, L. Liu, B. Ding, and H. Wang. 2011. Distance-constraint reachability computation in uncertain graphs. *PVLDB* 4, 9 (2011), 551–562.

[21] D. Kempe, J. Kleinberg, and É. Tardos. 2003. Maximizing the spread of influence through a social network. In *KDD'03*. https://doi.org/10.1145/956750.956769

[22] A. Khan, F. Bonchi, A. Gionis, and F. Gullo. 2014. Fast Reliability Search in Uncertain Graphs.. In *Proc. EDBT*. 535–546.

[23] A. Khan, F. Bonchi, F. Gullo, and A. Nufer. 2018. Conditional Reliability in Uncertain Graphs. *IEEE Trans. Knowl. Data Eng.* (2018).

[24] W. Khaouid, M. Barsky, V. Srinivasan, and A. Thomo. 2015. K-core decomposition of large networks on a single PC. In *Proc. VLDB*. ACM, 13–23.

[25] H. Kobayashi, B.L. Mark, and W. Turin. 2011. *Probability, Random Processes, and Statistical Analysis*. Cambridge University Press.

[26] G. Kollios, M. Potamias, and E. Terzi. 2013. Clustering large probabilistic graphs. *TKDE* (2013).

[27] N. Korovaiko and A. Thomo. 2013. Trust prediction from user-item ratings. *SNAM* 3, 3 (2013), 749–759.

[28] U. Kuter and J. Golbeck. 2010. Using probabilistic confidence models for trust inference in web-based social networks. *TOIT* 10, 2 (2010), 8.

[29] V. Lee, N. Ruan, R. Jin, and Ch. Aggarwal. 2010. A survey of algorithms for dense subgraph discovery. In *Managing and Mining Graph Data*. Springer, 303–336.

[30] X. Li, M. Wu, Ch. Kwoh, and S. Ng. 2010. Computational approaches for detecting protein complexes from protein interaction networks: a survey. *BMC genomics* 11, 1 (2010), S3.

[31] L. Liu, R. Jin, Ch. Aggarwal, and Y. Shen. 2012. Reliable clustering on uncertain graphs. In *Proc. ICDM*. IEEE, 459–468.

[32] Y. Liu, J. Lu, H. Yang, X. Xiao, and Zh. Wei. 2015. Towards maximum independent sets on massive graphs. *PVLDB* 8, 13 (2015), 2122–2133.

[33] A. Lyapunov. 1900. Sur une proposition de la théorie des probabilités. *Bulletin de l'Academie Impériale des Sci. de St. Petersbourg* 13 (1900), 359-386.

[34] A. Lyapunov. 1901. Nouvelle forme de la théoreme dur la limite de probabilité. *Mémoires de lâĂŹAcademie Impériale des Sci. de St. Petersbourg* 12 (1901), 1-24.

[35] A. Montresor, F. De Pellegrini, and D. Miorandi. 2013. Distributed k-core decomposition. *IEEE Trans. Parallel Distrib. Syst.* 24, 2 (2013), 288-300.

[36] A. P. Mukherjee, P. Xu, and S. Tirthapura. 2015. Mining maximal cliques from an uncertain graph. In *Proc. ICDE*. IEEE, 243–254.

[37] O. Papapetrou, E. Ioannou, and D. Skoutas. 2011. Efficient discovery of frequent subgraph patterns in uncertain graph databases. In *Proc. EDT*. ACM, 355–366.

[38] M. Potamias, F. Bonchi, A. Gionis, and G. Kollios. 2010. K-nearest neighbors in uncertain graphs. *PVLDB* 3, 1-2 (2010), 997–1008.

[39] A. E. Sarıyüce, B. Gedik, G. Jacques-Silva, K. Wu, and Ü. V. Çatalyürek. 2016. Incremental k-core decomposition: algorithms and evaluation. *VLDB* 25, 3 (2016), 425–447.

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

[41] R. Sharan, I. Ulitsky, and R. Shamir. 2007. Network-based prediction of protein function. *Mol. Syst. Biol.* 3, 1 (2007), 88.

[42] I. G. Shevtsova. 2010. An improvement of convergence rate estimates in the Lyapunov theorem. *Doklady Mathematics* 82, 3 (2010), 862-864.

[43] Y. Tang, X. Xiao, and Y. Shi. 2014. Influence Maximization: Near-Optimal Time Complexity Meets Practical Efficiency. *CoRR* abs/1404.0900 (2014).

[44] J. Ugander, L. Backstrom, C. Marlow, and J. Kleinberg. 2012. Structural diversity in social contagion. *Proc. Natl. Acad. Sci.* 109, 16 (2012), 5962–5966.

[45] D. Wen, L. Qin, Y. Zhang, X. Lin, and J. Yu. 2016. I/O efficient core graph decomposition at web scale. In *Proc. ICDE*. IEEE, 133–144.

[46] T. Wolf, A. Schröter, D. Damian, L. D. Panjer, and T. H. Nguyen. 2009. Mining task-based social networks to explore collaboration in software teams. *IEEE Soft.* 26, 1 (2009), 58–66.

[47] H. Wu, J. Cheng, Y. Lu, Y. Ke, Y. Huang, D. Yan, and H. Wu. 2015. Core decomposition in large temporal graphs. In *Proc. Big Data*. IEEE, 649–658.

[48] Y. Yuan, L. Chen, and G. Wang. 2010. Efficiently answering probability threshold-based shortest path queries over uncertain graphs. In *Proc. DSFAA*. Springer, 155–170.

[49] Y. Yuan, G. Wang, L. Chen, and H. Wang. 2012. Efficient subgraph similarity search on large probabilistic graph databases. *PVLDB* 5, 9 (2012), 800–811.

[50] Zh. Zhang, J. Yu, L. Qin, L. Chang, and X. Lin. 2015. I/O efficient: computing SCCs in massive graphs. *VLDB* 24, 2 (2015), 245–270.

[51] Zh. Zhang, J. Yu, L. Qin, and Z. Shang. 2015. Divide & conquer: I/O efficient depth-first search. In *Proc. SIGMOD*. ACM, 445–458.

[52] Zh. Zou, H. Gao, and J. Li. 2010. Discovering frequent subgraphs over uncertain graph databases under probabilistic semantics. In *Proc. SIGKDD*. ACM, 633–642.

[53] Zh. Zou, J. Li, H. Gao, and Sh. Zhang. 2010. Finding top-k maximal cliques in an uncertain graph. In *Proc. ICDE*. IEEE, 649–652.

[54] Z. Zou, J. Li, H. Gao, and S. Zhang. 2010. Mining frequent subgraph patterns from uncertain graph data. *IEEE Trans. Knowl. Data Eng.* 22, 9 (2010), 1203–1218.

[55] Zh. Zou and R. Zhu. 2017. Truss decomposition of uncertain graphs. *KAIS* 50, 1 (2017), 197–230.

[56] D. Zwillinger and S. Kokoska. 2000. *CRC Standard probability and statistics tables and formulae*. Chapman and Hall/CRC, USA.

[57] D. Zwillinger and S. Kokoska. 2013. *Probability Theory*. Springer-Verlag, London.