Efficient k-Clique Counting on Large Graphs: The Power of Color-Based Sampling Approaches

Xiaowei Ye, Rong-Hua Li, Qiangqiang Dai, Hongzhi Chen, and Guoren Wang

***Abstract*—K-clique counting is a fundamental problem in network analysis which has attracted much attention in recent years. Computing the count of k-cliques in a graph for a large k (e.g., $k = 8$) is often intractable as the number of k-cliques increases exponentially w.r.t. (with respect to) k. Existing exact k-clique counting algorithms are often hard to handle large dense graphs, while sampling-based solutions either require a huge number of samples or consume very high storage space to achieve a satisfactory accuracy. To overcome these limitations, we propose a new framework to estimate the number of k-cliques which integrates both the exact k-clique counting technique and three novel color-based sampling techniques. The key insight of our framework is that we only apply the exact algorithm to compute the k-clique counts in the *sparse regions* of a graph, and use the proposed color-based sampling approaches to estimate the number of k-cliques in the *dense regions* of the graph. Specifically, we develop three novel dynamic programming based k-color set sampling techniques to efficiently estimate the k-clique counts, where a k-color set contains k nodes with k different colors. Since a k-color set is often a good approximation of a k-clique in the dense regions of a graph, our sampling-based solutions are extremely efficient and accurate. Moreover, the proposed sampling techniques are space efficient which use near-linear space w.r.t. graph size. We conduct extensive experiments to evaluate our algorithms using 8 real-life graphs. The results show that our best algorithm is at least one order of magnitude faster than the state-of-the-art sampling-based solutions (with the same relative error 0.1%) and can be up to three orders of magnitude faster than the state-of-the-art exact algorithm on large graphs.**

***Index Terms*—k-clique counting, cohesive subgraphs, dynamic programming, graph coloring, graph sampling.**

I. Introduction

REAL-LIFE networks, such as social networks, web graphs, and biological networks, often contain frequently-occurring small subgraph structures. Such frequent small subgraphs are referred to as network motifs [1]. Counting the motifs is a fundamental tool in many network analysis applications, including social network analysis, community detection, and bioinformatics [1], [2], [3], [4], [5]. Perhaps the most elementary motif in a graph is the k-clique which has been widely used in a variety of network analysis applications [1], [2], [6], [7], [8].

Given a graph G, a k-clique is a complete subgraph of G with k nodes. Counting the k-cliques in a graph has found many important applications in dense subgraph mining and social network analysis. For example, Sariyüce et al. [9] proposed a nucleus decomposition method to find the hierarchy of dense subgraphs, which uses the k-clique counting operator as a basic building block. Tsourakakis [8] studied a k-clique densest subgraph problem which also uses the k-clique counting operator as a building block. Additionally, the k-clique counting operator has also been applied to detect higher-order organizations in social networks [10], [11].

Motivated by the above applications, many practical k-clique counting algorithms have been proposed [12], [13], [14], [15], [16], [17], [18], [19], [20]. Existing k-clique counting algorithms can be classified into (1) exact k-clique counting methods, and (2) sampling-based approximation solutions. Chiba and Nishizeki [12] developed the first exact k-clique counting algorithm based on k-clique enumeration which is very efficient on real-life sparse graphs for a small k. Such an algorithm was recently improved by Finocchi et al. [13] based on a degree ordering technique. Subsequently, Danisch et al. [15] further improved this algorithm by using a degeneracy ordering technique [21]. More recently, Li et al. [16] developed a further improved algorithm based on a hybrid of degeneracy and color ordering technique. All these exact k-clique counting algorithms are based on k-clique enumeration, which are typically intractable on large graphs for a large k (e.g., $k \geq 8$) due to combinatorial explosion. To overcome this issue, Jain and Seshadhri developed an elegant algorithm, called PIVOTER, based on a classic pivoting technique which was widely used for pruning the search branches in maximal clique enumeration [22]. The key idea of PIVOTER is that it can implicitly construct a succinct clique tree (SCT) by using the pivoting technique in the search procedure. Such a SCT structure maintains a unique representation of all k-cliques, but its size is much smaller than the number of k-cliques. PIVOTERwas shown to be much faster than previous k-clique enumeration based algorithms [12], [13], [14], [15], [16]. Although PIVOTER is often very efficient for handling real-life sparse graphs, it may still have a very deep recursion tree when processing the dense regions of the graph, which is the main bottleneck of the PIVOTER algorithm. Moreover, PIVOTER is based on the idea of enumeration of large

Manuscript received 9 November 2022; revised 26 July 2023; accepted 26 August 2023. Date of publication 12 September 2023; date of current version 8 March 2024. This work was supported in part by the National Key Research and Development Program of China under Grant 2020AAA0108503, in part by the NSFC under Grants U2241211 and 62072034, and in part by CCF-Huawei Populus Grove Fund. Recommended for acceptance by A. Khan. *(Corresponding author: Rong-Hua Li.)*

Xiaowei Ye, Rong-Hua Li, Qiangqiang Dai, and Guoren Wang are with the Beijing Institute of Technology, Beijing 100811, China (e-mail: yexiaowei@bit.edu.cn; lironghuabit@126.com; qiangd66@gmail.com; wanggrbit@126.com).

Hongzhi Chen is with ByteDance, Beijing 100811, China (e-mail: chenhongzhi@bytedance.com).

Digital Object Identifier 10.1109/TKDE.2023.3314643

cliques (not necessary maximal cliques) to count the k-cliques. It is often not very fast on the dense regions of the graph, because the dense regions of the graph may contain many large cliques (with complicated overlap relationships), resulting in a large search tree of PIVOTER (e.g., see the results on the LiveJournal dataset in [17]).

Approximation solutions based on sampling are typically able to handle large dense graphs when k is not very large [18], [20], [23]. However, to achieve a desired accuracy, previous sampling-based solutions either require a huge number of samples [18], [24], [25] or consume very high storage space [19], [20], [23], [26], [27] for a relatively large k (e.g., $k \geq 8$). Among them, a notable sampling-based approximation algorithm is the TuranShadow algorithm which was proposed by Jain and Seshadhri [20]. As shown in [20], TuranShadow is much faster and more accurate than the other previous sampling-based algorithms. The main limitation of TuranShadow is that it needs to take $O(n\alpha^{(k-1)} + m)$ time and $O(n\alpha^{(k-2)})$ space to construct a data structure called Tuŕan Shadow for sampling, where α denotes the arboricity of the graph [12]. Therefore, on large graphs, TuranShadow is very costly for a large k. To reduce the space usage of TuranShadow, the same authors developed an improved TuranShadow algorithm called PEANUTS. PEANUTS adopts an online sampling solution which does not construct the Tuŕan Shadow offline. However, PEANUTS still needs to build a *partial* Tuŕan Shadow when estimating the k-clique counts of a sampled node, which sometimes consumes a lot of space.

To overcome the limitations of the state-of-the-art algorithms, we propose a new framework to estimate the number of k-cliques in a graph which integrates both the exact PIVOTER algorithm and two newly-developed sampling-based techniques. Our framework is based on a simple but effective observation: PIVOTER is extremely efficient to compute the number of k-cliques in the *sparse regions* of the graph, while sampling-based solutions are often very efficient and accurate to estimate the k-clique counts in the *dense regions* of a graph. Base on this crucial observation, we can first partition the graph into sparse and dense regions. Then, for the sparse regions, we invoke PIVOTER to exactly compute the k-clique counts. For the dense regions, we propose three novel sampling techniques based on a concept of graph coloring [28] to estimate the k-clique counts. Specifically, we first present a new concept called k-color set which denotes a set of k nodes with k different colors. Then, we propose a dynamic programming (DP) based k-color set sampling algorithm to estimate the k-clique counts. Since a k-color set is typically a good approximation for a k-clique in the dense regions of a graph, our algorithm is extremely efficient and accurate. In addition, we also propose a novel DP-based k-color path sampling and a novel DP-based k-triangle path sampling techniques to further improve the efficiency and accuracy. Here a k-color path is a connected k-color set and a k-triangle path is a k-color path with any three consecutive nodes forming a triangle. These two new concepts are more effective to approximate a k-clique than the k-color set. Moreover, unlike TuranShadowand PEANUTS, all of our sampling-based solutions take near-linear space w.r.t. the graph size.

Contributions: In summary, the main contributions of this paper are as follows.

- We propose a new algorithmic framework for estimating k-clique counting which can circumvent the defects of the existing exact and approximation algorithms. We show that our framework is extremely efficient and accurate. It can achieve a 10^{-5} relative error by sampling a reasonable number of samples.
- We develop three novel DP-based k-color set sampling techniques to estimate the number of k-cliques in the dense regions of the graph. Our novelty is in the algorithmic use of classic graph coloring technique for sampling. The striking features of our techniques are that they are not only very efficient and accurate, but also use near-linear space w.r.t. the graph size.
- We evaluate our algorithms on 8 large real-life graphs. The results show that (1) our best algorithm is at least one order of magnitude faster than the state-of-the-art approximate algorithm (PEANUTS) to achieve a 0.1% relative error, using much smaller space; and (2) it can be up to three orders of magnitude faster than the state-of-the-art exact algorithm (PIVOTER) on large graphs. For example, on the hardest dataset LiveJournal with $k = 8$, TuranShadow takes more than 120 seconds and PIVOTER cannot terminate within 5 hours, while our best algorithm consumes around 20 seconds to achieve a 0.1% relative error. Moreover, our algorithms also exhibit an excellent parallel performance which can achieve $12\times \sim 14\times$ speedup ratios when using 16 threads in our experiments.

Reproducibility: For reproducibility purpose, the source code of this paper is released at https://github.com/LightWant/dpcolor.

Organization: The rest of this paper is organized as follows. In Section II, we describe several key notations, formulate the problem, summarize several representative existing algorithms of k-clique counting, and also analyze the defects of these algorithms. In Section III, we propose a novel sampling framework for k-clique counting. In Section IV, we present the DP-based k-color set sampling algorithm. The k-color path and k-triangle path algorithms are developed in Sections V and VI respectively. Extensive experiments are shown in Section VIII. Finally, we survey the related work in Section IX and conclude this work in Section X.

II. Preliminaries

Let $G = (V, E)$ be an undirected graph, where V and E denotes the set of nodes and edges respectively. Let n and m be the number of nodes and edges of G respectively. Denote by $N_v(G)$ the set of neighbors of v in G. The degree of v, denoted by $d_v(G)$, is the size of the neighbor set of v, i.e., $d_v(G) = |N_v(G)|$. Given a subset S of V, we denote by $G(S) = (V_S, E_S)$ the subgraph of G induced by S, where $E_S = \{(u, v) \in E | u, v \in S\}$. A k-clique is a complete subgraph of G in which every pair of nodes is connected by an edge.

Given a graph G and an integer k, the k-clique counting problem is to compute the number of k-cliques in G. Practical algorithms for solving the k-clique counting problem are often based on some ordering-based heuristic techniques [15], [16], [17], [20].

Algorithm 1: The PIVOTER Algorithm [17].

Input: A graph $G = (V, E)$ and an integer k
Output: The number of k-cliques in G
1 $\vec{G} \leftarrow$ the DAG generated by the degeneracy ordering of G;
2 $ans \leftarrow 0$;
3 **for** $u \in V$ **do**
4 PIVOTER$(N_u(\vec{G}), k-1, 0, 0)$;
5 **return** ans;
6 **Procedure** PIVOTER(S, k, p, h)
7 **if** $h > k$ **return**;
8 **if** $S = \emptyset$ **then**
9 $ans \leftarrow ans + \binom{p}{k-h}$;
10 **return**;
11 $pv \leftarrow \max_{u \in S}\{|N_u(\vec{G}) \cap S|\}$;
12 PIVOTER$(N_{pv}(\vec{G}) \cap S, k, p+1, h)$;
13 $U \leftarrow S - N_{pv}(\vec{G}) - \{pv\}$;
14 **for** $v_i \in U$ **do**
15 PIVOTER$(N_{v_i}(\vec{G}) \cap S, k, p, h+1)$;
16 $S \leftarrow S - \{v_i\}$;

Let $\pi : V \rightarrow \{v_1, \ldots, v_n\}$ be a total order of the nodes in G. For two nodes u and v of G, we say that $\pi(v) < \pi(u)$ if u comes after v in the ordering of π. Then, based on such an ordering, we can obtain a DAG (directed acyclic graph) $\vec{G}$ by orienting the edges of the undirected graph G. Specifically, for each undirected edge (u, v) in G, we obtain a directed edge (u, v) in $\vec{G}$ if $\pi(u) < \pi(v)$, otherwise we get a directed edge (v, u). The k-clique counting problem in G is equivalent to computing the number of k-cliques in $\vec{G}$. Existing k-clique counting algorithms that work on the DAG $\vec{G}$ (instead of the original graph G) can guarantee that each k-clique is only explored once, thus significantly improving the efficiency.

Note that many different ordering heuristics for k-clique counting have been developed in the literature [16]. Among them, a widely-used ordering heuristics is the degeneracy ordering [21], where the degeneracy is a metric to measure the sparsity of a graph [29]. Specifically, the degeneracy ordering of nodes in G is defined as an ordering $\{v_1, \ldots, v_n\}$ such that the degree of v_i is minimum in the subgraph of G induced by $\{v_i, \ldots, v_n\}$ for each v_i in G. We can make use of a classic peeling algorithm to generate the degeneracy ordering in $O(m + n)$ time [30]. Let δ be the degeneracy of G. Then, we can easily derive that $d_v(\vec{G}) \leq \delta$. Since δ is often very small in real-world graphs [21], [29], the degeneracy ordering based k-clique counting algorithms are often very efficient in practice [16]. In this work, we will also use the degeneracy ordering to design our algorithms.

A. Existing Algorithms and Their Limitations

The PIVOTER algorithm: PIVOTER is the state-of-the-art exact k-clique counting algorithm which was proposed by Jain and Seshadhri [17]. The PIVOTER algorithm is based on a classic pivoting technique which has been widely used for pruning the search branches in maximal clique enumeration [22]. The key idea of PIVOTER is that it implicitly builds a succinct clique tree (SCT) by using the pivoting technique in the search procedure. Such a SCT structure maintains a unique representation of all k-cliques, but its size is often much smaller than the number of k-cliques. PIVOTER was shown to be much faster than the traditional k-clique listing based algorithms [15], [16], [17]. Since we will make use of PIVOTER as a subroutine in our algorithms, we give the detailed description of PIVOTER in Algorithm 1.

Algorithm 1 first computes a DAG $\vec{G}$ of G based on the degeneracy ordering (line 1). Then, for each node $u \in V$, the algorithm invokes the PIVOTER procedure to calculate the number of $(k-1)$-cliques in $N_u(\vec{G})$ (lines 3-4). In the PIVOTER procedure, it first selects a node with the maximum number of neighbors in S as a pivot node pv (line 11). The candidate set S is then divided into three subsets: $\{pv\}$, $N_{pv}(\vec{G}) \cap S$ and $S - \{pv\} - N_{pv}(\vec{G})$. By these three subsets, the cliques can be classified into three various types: (1) the k-cliques containing nodes in both $\{pv\}$ and $N_{pv}(\vec{G}) \cap S$, (2) the k-cliques only containing nodes in $N_{pv}(\vec{G}) \cap S$, and (3) the k-cliques containing nodes in $S - \{pv\} - N_{pv}(\vec{G})$. Then, PIVOTER recursively computes the total numbers for these three types of k-cliques (lines 12-16). Note that the first two types of k-cliques can be counted by invoking PIVOTER with the input set $N_{pv}(\vec{G}) \cap S$ (line 12), whereas the last type of k-cliques are iteratively counted for each node in $S - \{pv\} - N_{pv}(\vec{G})$ (lines 14-16). The worst-case time complexity of PIVOTER is $O(n\alpha 3^{\alpha/3})$ where α is the arboricity [12] of the graph and $\delta/2 \leq \alpha \leq \delta$. Since α is often very small in real-life sparse graphs, the PIVOTER algorithm was shown to be very efficient in practice [17].

The TuranShadow algorithm and its variant: TuranShadow is a representative sampling-based approximation algorithm which was also proposed by Jain and Seshadhri [20]. As shown in [20], TuranShadow is much faster and more accurate than the other sampling-based algorithms. The TuranShadow algorithm first constructs a data structure, called Tuŕan Shadow, based on the classic Tuŕan's theorem which states that a graph must contain a k-cliuqe if the edge density $\rho = m/\binom{n}{2}$ satisfies $\rho > 1 - 1/(k-1)$. Specifically, the Tuŕan Shadow, denoted by $\mathcal{S}$, contains a set of pairs (S, l) where S is a node set and $l \leq k$ is an integer. Let G_S be the subgraph induced by the node set S. For each pairs (S, l), the edge density of G_S is larger than $1 - 1/(l-1)$, thus G_S must contain an l-clique by Tuŕan's theorem. Jain and Seshadhri [20] showed that there is a one-to-one mapping between a k-clique in G and an l-clique in G_S for a pair (S, l) in $\mathcal{S}$. Therefore, to count the number of k-cliques, it is sufficient to calculate the number of l-cliques in G_S for each pair (S, l), which can be efficiently estimated by a weighted sampling procedure [20]. In [20], Jain and Seshadhri also developed an algorithm with $O(\alpha|\mathcal{S}| + m)$ time complexity to construct the Tuŕan Shadow, where α is the arboricity of the graph and $|\mathcal{S}| = O(n\alpha^{(k-2)})$. Since α is typically very small in real-life graphs, TuranShadow is efficient to estimate the k-clique counts. Recently, Jain and Seshadhri proposed an improved Tuŕan Shadow algorithm, namely PEANUTS [27], which can be considered as the state-of-the-art sampling-based approximation algorithm. PEANUTS does not construct the

Algorithm 2: The Proposed Framework.

Input: A graph $G = (V, E)$, an integer k, and the sample size t
Output: The number of k-cliques in G
1 $\vec{G} \leftarrow$ the DAG generated by the degeneracy ordering of G;
2 $ans \leftarrow 0$; $S \leftarrow \emptyset$;
3 **foreach** $v \in V$ **do**
4 **if** $\bar{d}(G(N_v(\vec{G}))) < k$ **then** $ans \leftarrow ans + \mathsf{PIVOTER}(N_v(\vec{G}), k-1)$;
5 **else** $S \leftarrow S \cup \{v\}$;
6 **return** $ans + \mathsf{Sampling}(\vec{G}, S, k, t)$;

Tuŕan Shadow offline. Instead, it builds a *partial* Tuŕan Shadow for a sampled node during the sampling procedure, thus it uses much less space than the original TuranShadow algorithm. Moreover, PEANUTS is often much faster than TuranShadow, since it is no need to construct the whole Tuŕan Shadow which takes much time in the original TuranShadow algorithm.

Limitations of the state-of-the-art algorithms: Although the PIVOTER algorithm is often very efficient for handling real-life sparse graphs (because real-life graphs often have a small arboricity), it is still intractable when processing some hard instances, such as the LiveJournal graph in [17]. The reason may be that such hard instances often have a huge number of maximal cliques, thus the succinct clique tree (SCT) of the PIVOTER algorithm can be very large, rendering the algorithm intractable. TuranShadow is generally faster than the exact PIVOTER algorithm for handling dense graphs with a provably small relative error. However, the main limitations of TuranShadoware twofold: (1) it uses $O(n\alpha^{(k-2)})$ space to store the Tuŕan Shadow which is very costly for large graphs; and (2) it often needs to take much time to construct the Tuŕan Shadow for large graphs (the construction time is $O(n\alpha^{(k-1)} + m)$). Such two limitations are alleviated by the improved Tuŕan Shadow algorithm PEANUTS. However, on some large graphs, PEANUTS needs to take much time to construct the partial Tuŕan Shadow and uses considerable space, thus it is still not very efficient when processing large graphs.

III. The Proposed Framework

In this section, we propose a new algorithmic framework to estimate the number of k-cliques which combines both the exact PIVOTER algorithm and the sampling-based algorithms. The key idea of our framework is based on a simple but effective observation. The PIVOTER algorithm often works very efficient in the *sparse regions* of the graph, in which the number of k-cliques is typically not very large. However, in the *dense regions* of the graph, PIVOTER may be very costly to compute the k-clique counts, as the dense regions of the graph may contain a huge number of k-cliques. On the contrary, the sampling-based solutions are often very efficient and accurate to estimate the number of k-cliques in the dense regions of the graph, but they generally perform very bad in the sparse regions of the graph. This is because the k-cliques are relatively easier to be sampled in the dense regions, but they are often very hard to be drawn from the sparse regions of the graph. Therefore, to overcome the limitations of both the exact and sampling algorithms, we can apply the exact PIVOTER algorithm to calculate the k-clique counts in the sparse regions of the graph, and use the sampling-based techniques to estimate the number of k-cliques in the remaining dense regions of the graph. The details of our framework is shown in Algorithm 2.

Note that in Algorithm 2, we make use of the average degree of the nodes in the subgraph $C = (V_C, E_C)$ of G, denoted by $\bar{d}(V_C) = \sum_{v \in V_C} d_v(C)/|V_C|$, as an indicator to measure the sparsity of C. We refer to a subgraph C of G as a dense subgraph of G if $\bar{d}(V_C) \geq k$ (i.e., it lies in the dense regions of G), otherwise it is called a sparse subgraph. In Algorithm 2, it first computes a DAG $\vec{G}$ by the degeneracy ordering of G (line 1). Let $N_v(\vec{G})$ be the out-neighbors of a node v in $\vec{G}$, and $G(N_v(\vec{G}))$ be the subgraph induced by $N_v(\vec{G})$ in G. If the average degree of $G(N_v(\vec{G}))$ is smaller than k, the algorithm invokes PIVOTER to exactly compute the number of $(k-1)$-cliques contained in $N_v(\vec{G})$ (line 4). Otherwise, the subgraph $G(N_v(\vec{G}))$ is considered as a dense region of G, and the $(k-1)$-cliques contained in $N_v(\vec{G})$ are estimated by a sampling algorithm (lines 5-6). Let α be the arboricity [12] of the input graph $G = (V, E)$ and V' be the set of nodes in the sparse region of G. Then, we have the following result.

Theorem 1: The time complexity of Line 4 of Algorithm 2 is $O(|V'|\alpha\lceil\sqrt{k\alpha + \frac{1}{2}}\rceil 3^{\frac{\lceil\sqrt{k\alpha+\frac{1}{2}}\rceil}{3}})$.

Proof: For a node v in V', we use the notion α_v to denote the arboricity of $G(N_v(\vec{G}))$, m_v to denote the count of edges in $G(N_v(\vec{G}))$, and n_v to denote the count of nodes $|N_v(\vec{G})|$. It is easy to derive that $n_v \leq \delta \leq 2\alpha$ according to the degeneracy ordering. By $\bar{d}(G(N_v(\vec{G}))) < k$ (Line 4 of Algorithm 2), we can derive that $m_v = |N_v(\vec{G})| \times \bar{d}(G(N_v(\vec{G}))) < k|N_v(\vec{G})| \leq 2k\alpha$. Then, we have $\alpha_v \leq \lceil\frac{\sqrt{2m_v+n_v}}{2}\rceil \leq \lceil\sqrt{k\alpha + \frac{1}{2}}\rceil$ [12]. Thus, the total time complexity is $O(\sum_{v \in V'} |N_v(\vec{G})|\alpha_v 3^{\alpha_v/3})$, because the time complexity of PIVOTER is $O(n\alpha 3^{\alpha/3})$ for a graph with n nodes and arboricity α [17]. As a result, we can derive that the time complexity of Line 4 of Algorithm 2 is $O(|V'|\alpha\lceil\sqrt{k\alpha + \frac{1}{2}}\rceil 3^{\frac{\lceil\sqrt{k\alpha+\frac{1}{2}}\rceil}{3}})$.□

Note that Theorem 1 shows the time complexity of Algorithm 2 in the sparse region of the graph. Since $\lceil\sqrt{k\alpha + \frac{1}{2}}\rceil$ is smaller than α (because k is usually very small), our framework is efficient on the sparse region of the graphs.

The remaining question is how can we devise an efficient and effective sampling algorithm to estimate the number of k-cliques in the dense regions of G. Traditional edge sampling algorithms, such as [18], [31], are often inefficient, because those algorithms require a considerable number of samples to achieve a desired accuracy [20]. The color-coding based techniques often consume a significant number of space [19], [23], [26] and also they are less efficient than the TuranShadow algorithm [20]. The TuranShadow algorithm and its variant [20], [27], which are the state-of-the-art sampling-based techniques, also need

much space to store the (paritial) Tuŕan Shadow. Moreover, the construction time of the (partial) Tuŕan Shadow is often very long for large graphs, because the worst-case time complexity of TuranShadow is exponential. In Sections IV, V and VI, we will propose three novel and efficient sampling algorithms to tackle this problem.

Parallel implementation: Note that the proposed framework (Algorithm 2) can be easily parallelized, because the number of k-cliques in the subgraph induced by the out-neighbors for each node in $\vec{G}$ is independent. Specifically, in lines 3-5 of Algorithm 2, we can process the nodes in the sparse regions in parallel by independently invoking the PIVOTER algorithms. In the dense regions, the sampling-based techniques are also easily to be parallelized, because we can always draw t independent samples in parallel. In our experiments, we will show that our parallel implementations can achieve a near-linear speedup ratio on real-life graphs.

IV. K-Color Set Sampling

In this section, we develop a novel sampling approach to estimate the k-clique counts in the dense regions of the graph, called k-color set sampling. Our technique is based on a concept of graph coloring [28], [32], [33]. Specifically, we first color the nodes in a graph such that each pair of adjacent nodes are colored with different colors. Let χ be the number of colors that are used to color all nodes in the graph G. The graph coloring procedure assigns an integer color value taking from $[1, \ldots, \chi]$ to each node in G, and no two adjacent nodes have the same color value. Note that since the minimum coloring problem (χ is minimum) is NP-hard [28], we use a linear-time greedy coloring algorithm [32], [33] to obtain a feasible coloring solution. Based on a feasible coloring solution, we define a concept called k-color set as follows.

Definition 1: A set of nodes V_k in the colored graph G is called a k-color set if it contains k nodes with k different colors.

Note that by Definition 1, the nodes of any k-clique must form a k-color set. In particular, we have the following lemma.

Lemma 1: Given a graph G, all k-cliques must be contained in the set of all k-color sets.

Let $\mathsf{cnt}_k(G, clique)$ and $\mathsf{cnt}_k(G, color)$ be the number of k-cliques and k-color sets of G respectively. Denoted by ρ_c the k-clique density of a graph G which is defined as the ratio between the number of k-cliques and the number of k-color sets of G, i.e., $\rho_c = \frac{\mathsf{cnt}_k(G,clique)}{\mathsf{cnt}_k(G,color)}$. Intuitively, in the dense regions of the graph G, a k-color set is *likely* to be a k-clique. Therefore, the k-clique density ρ_c of the dense region of G is often not very small. As a consequence, an effective sampling technique to estimate the number of k-cliques can be obtained by estimating ρ_c.

There are two nontrivial problems needed to be tackled to develop such a sampling technique. First, we need to devise an efficient algorithm to compute the number of k-color sets. Second, to estimate ρ_c, we need to develop a uniform sampling mechanism to sample the k-color sets. Below, we will propose a dynamic programming algorithm to solve these issues.

A. DP-Based k-Color Set Sampling

Here we first propose a DP algorithm to compute the number of k-color sets. Then, we show how to use the DP algorithm to uniformly sample a k-color set.

Counting the number of k-color sets: Let χ be the number of colors of the graph G obtained by the greedy coloring algorithm [32], [33]. Denote by a_i the number of nodes in G with the color $i \in [1, \chi]$. Let G_i be the subgraph of G that only contains the nodes of G with color values no larger than i, i.e., $G_i = (V_i, E_i)$, where $V_i = \{v \in V | c(v) \leq i\}$, $E_i = \{(u, v) \in E | u, v \in V_i\}$, and $c(v)$ is the color value of v in G. Let $F(i, j)$ be the number of j-color sets in G_i. Then, we have the following recursive function for all $i, j \in [1, \chi]$.

$$F(i, j) = a_i \times F(i - 1, j - 1) + F(i - 1, j). \tag{1}$$

The key idea of (1) is that the number of j-color sets in G_i can be derived by considering two cases: (1) the color i is included in the j-color sets; and (2) the color i is not included in the j-color sets. For the first case, the number of j-color sets in G_i is equal to a_i times the number of $(j-1)$-color sets in G_{i-1}, i.e., $a_i \times F(i-1, j-1)$. For the second case, the number of j-color sets is equal to the number of j-color sets in G_{i-1}, which is $F(i-1, j)$. Thus, the total number of j-color sets in G_i is the sum over these two cases. Clearly, the number of k-color sets in G is equal to the number of k-color sets in G_χ, i.e., $F(\chi, k)$. In addition, the initial states of $F(i, j)$ are as follows:

$$\begin{cases} F(i, 0) = 1, & \text{for all } i \in [0, \chi], \\ F(i, j) = 0, & \text{for all } i \in [0, \chi], j \in [i + 1, \chi]. \end{cases} \tag{2}$$

Based on (1) and (2), we can compute the number of k-color sets $F(\chi, k)$ in $O(k\chi)$ time by dynamic programming. The detailed implementation of the DP algorithm can be found in the DPCount procedure of Algorithm 3 (see lines 5-12).

From counting to uniformly sampling: Here we propose an efficient approach to uniformly sample a k-color set based on the k-color set counting technique. For convenience, we refer to a set of k different colors selected from $[1, \chi]$ as a k-color class. Clearly, in a graph G, a k-color class may contain a set of k-color sets.

To generate a uniform k-color set, a potential method is that we first sample a k-color class, and then we randomly select a node in G with color i for each i in the sampled k-color class. The challenge of this method is that how can we sample the k-color class to guarantee that the resulting k-color set is uniformly generated. Obviously, the straightforward method that uniformly picks k different colors from $[1, \chi]$ is incorrect in our case. This is because the numbers of k-color sets contained in various k-color classes are different. Thus, uniformly sampling a k-color class from $[1, \chi]$ will introduce biases for generating a uniform k-color set.

To overcome this challenge, we propose a DP algorithm to sample a k-color class which can guarantee that the resulting k-color set is uniformly drawn. In particular, given a j-color class in G_i, it either (1) contains the color i, or (2) does not contain the color i. If the first case is true, the other $j - 1$ colors of the

Algorithm 3: DPSampler(G, χ, k).

Input: A colored graph $G = (V, E)$, an integer k, and the maximum color number χ
Output: A uniformly sampled k-color set

1 $F \leftarrow$ DPCount(G, χ, k);
2 $p_{(i,j)} \leftarrow \frac{a_i \times F(i-1,j-1)}{F(i,j)}$ for all $i \in [1, \chi]$ and $j \in [1, k]$;
3 $R \leftarrow$ DPSampling$(G, P, \emptyset, \chi, k)$;
4 **return** R;
5 **Procedure** DPCount(G, χ, k)
6 Let a_i be the number of nodes with color i in G;
7 $F(i, j) \leftarrow 0$ for all $i \in [0, \chi]$ and $j \in [i + 1, k]$;
8 **foreach** $i = 0$ *to* χ **do** $F(i, 0) = 1$;
9 **foreach** $i = 1$ *to* χ **do**
10 **for** $j = 1$ *to* k **do**
11 $F(i, j) = a_i \times F(i - 1, j - 1) + F(i - 1, j)$;
12 **return** F;
13 **Procedure** DPSampling(G, P, R, i, j)
14 **if** $j = 0$ **then return** R;
15 Sampling the color i with probability $p_{(i,j)}$;
16 **if** *the color* i *is sampled* **then**
17 Randomly choose a node v in G with color i;
18 DPSampling $(G, P, R \cup \{v\}, i - 1, j - 1)$;
19 **else** DPSampling $(G, P, R, i - 1, j)$;

j-color class are selected from $[1, i-1]$ in G_{i-1}. However, for the second case, the j-color class must be selected from $[1, i-1]$ in G_{i-1}. Therefore, we can sample a k-color class in G based on a similar DP equation as shown in (1). More specifically, to sample a j-color class, we define the probability of selecting the color i in G_i as

$$p_{(i,j)} = \frac{a_i \times F(i-1, j-1)}{F(i, j)}. \tag{3}$$

Clearly, the probability that does not choose the color i in G_i is $1 - p_{(i,j)} = F(i-1, j)/F(i, j)$. Based on (3), we can sample a j-color class using the following recursive sampling procedure. In each recursion, we pick a color i in G_i with the probability $p_{(i,j)}$. If the color i is sampled, we recursively sample the $(j-1)$-color class in G_{i-1}. Otherwise, we recursively sample the j-color class in G_{i-1}. After obtaining a k-color class, a k-color set is generated by randomly selecting a node with each color i in the k-color class. The detailed implementation of our algorithm for uniformly sampling a k-color set is shown in Algorithm 3.

Algorithm 3 first invokes the DP procedure to compute $F(i, j)$ for every $i \in [1, \chi]$ and $j \in [1, k]$ (line 1 and lines 5-12). Then, the algorithm computes the probability $p_{(i,j)}$ based on (3) (line 2). After that, the algorithm calls the recursively sampling procedure to uniformly generate a k-color set (line 3 and lines 13-19). The following results ensure the correctness of Algorithm 3.

Lemma 2: The DPSampling procedure in Algorithm 3 outputs a k-color set of G if $\chi \geq k$.

Proof: On the one hand, it is easy to verify that there are at most k colors outputted by the DPSampling procedure, since $p_{(i,0)} = 0$ and DPSampling will terminate immediately when $j = 0$. On the other hand, by (3), we can derive that $p_{(i,i)} = 1$. This is because $F(i-1, i) = 0$ by definition, thus $1 - p_{(i,i)} = F(i-1, i)/F(i, i) = 0$. As a result, the probability of sampling a color i with $p_{(i,i)}$ is always 1, thus there are at least k colors that are sampled by DPSampling if $\chi \geq k$. Putting it all together, the lemma is established.□

Theorem 2: Algorithm 3 outputs a uniform k-color set.

Proof: Let X be the event of a random k-color class of G sampled by DPSampling. For each color j from 1 to χ, let Y_j be an indicator random variable, which is equal to 1 if the color j is selected in the event X, otherwise it is equal to 0. Let $\Pr(X)$ be the occurrence probability of the event X. Then, we have the following equation:

$$\Pr(X) = \Pr\left(\left(\sum_{i=1}^{\chi} Y_i\right) = k\right). \tag{4}$$

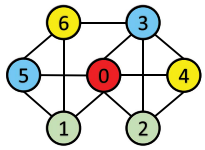
Recall that DPSampling draws k colors following the decreasing order of the color values (i.e., from χ to 1). For each color $j \in [1, \chi]$, the probability of selecting the color j in G_i is $p_{(i,j)}$. Assume that the sampled k-color class of G is $C = \{c_1, \ldots, c_k\}$, where each c_i is a color value of G and $c_1 > c_2 > \cdots > c_k$. Clearly, a k-color class C partitions the interval $[1, \chi]$ into at most $2k + 1$ sub-intervals as $\{[c_1 + 1, \chi], [c_1, c_1], [c_2 + 1, c_1 - 1], \ldots, [c_k + 1, c_{k-1} - 1], [c_k, c_k], [1, c_k - 1]\}$. Note that DPSampling only selects a color in the sub-intervals $[c_i, c_i]$ for every $i = 1, \ldots, k$, and no color is selected in the other sub-intervals. Therefore, the probability of $\Pr((\sum_{i=1}^{\chi} Y_i) = k)$ can be computed by

$$\begin{aligned}
&\frac{F(\chi-1,k)}{F(\chi,k)} \times \frac{F(\chi-2,k)}{F(\chi-1,k)} \times \cdots \times \frac{F(c_1,k)}{F(c_1+1,k)} \\
&\times \frac{a_{c_1} \times F(c_1-1,k-1)}{F(c_1,k)} \times \frac{F(c_1-2,k-1)}{F(c_1-1,k-1)} \times \cdots \\
&\times \frac{F(c_2,k-1)}{F(c_2+1,k-1)} \times \frac{a_{c_2} \times F(c_2-1,k-2)}{F(c_2,k-1)} \\
&\times \cdots \times \frac{F(c_k,1)}{F(c_k+1,1)} \times \frac{a_{c_k} \times F(c_k-1,0)}{F(c_k,1)} \\
&= \frac{a_{c_1} \times a_{c_2} \times \cdots \times a_{c_k}}{F(\chi,k)}.
\end{aligned} \tag{5}$$

After obtaining a k-color class C, the algorithm further samples k nodes with k different colors in C from G. Let $\Pr(k\text{-color set})$ be the probability of sampling a k-color set from G. Then, we have

$$\begin{aligned}
&\Pr(k\text{-color set}) \\
&= \Pr(\text{k nodes with different colors}|X) \times \Pr(X) \\
&= \frac{1}{a_{c_1} \times a_{c_2} \times \cdots \times a_{c_k}} \times \frac{a_{c_1} \times a_{c_2} \times \cdots \times a_{c_k}}{F(\chi,k)} \\
&= \frac{1}{F(\chi,k)} = \frac{1}{\mathsf{cnt}_k(G, color)}
\end{aligned} \tag{6}$$

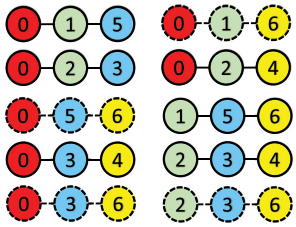
By (6), each k-color set is uniformly sampled, thus the theorem is established.□



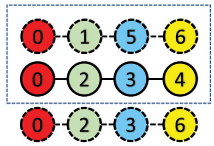
(a) An example graph

χ	k			
	1	2	3	4
1	1			
2	3	2		
3	5	8	4	
4	7	18	20	8

(b) The DP table of the k-color sets



(c) All 3-color paths.



(d) Three 4-color paths and two 4-triangle paths.

Fig. 1. Illustration of the three proposed color-based sampling techniques.

The following theorem shows the complexity of Algorithm 3.

Theorem 3: Suppose that the graph G is colored and the nodes in each color group are obtained. Then, both the time and space complexity of Algorithm 3 are $O(\chi k)$.

Proof: Clearly, the time complexity of the DP procedure for counting the number of k-color sets is $O(\chi k)$. In the DPSampling procedure, we can randomly choose a node with color i in constant time if the color groups are obtained (line 17). The total time costs of the DPSampling procedure are bounded by $O(\chi + k)$. As a result, the time complexity of Algorithm 3 is $O(\chi k)$. For the space complexity, Algorithm 3 only requires $O(\chi k)$ additional space to store the DP table F and the probabilities p.□

Example 1: Fig. 1(a) is a colored graph with $\chi = 4$. The color values of nodes $\{0, 1, 2, 3, 4, 5, 6\}$ is $\{1, 2, 2, 3, 4, 3, 4\}$, respectively. Clearly, we have $a_1 = 1$ and $a_i = 2$ for $i = 2, 3, 4$ respectively. Initially, we have $F(i, 0) = 1$ for all $i \in [0, 4]$, and $F(i, j) = 0$ for all $i \in [0, 4], j \in [i + 1, 4]$. By (1), we have $F(1, 1) = a_1 \times F(0, 0) + F(0, 1) = a_1 = 1$. $F(1, 1) = 1$, which means that there is only one way to choose a vertex with color 1. Similarly, we get $F(2, 1) = a_2 \times F(1, 0) + F(1, 1) = 3$, which means that there are 3 different ways to choose a vertex from the vertices with colors 1 and 2. The DP table is shown in Fig. 1(b). Then, we anlayze the probability of sampling the three nodes $\{0, 2, 3\}$ with color 1,2,3 respectively. Note that the probability of color 4 not being sampled is $1 - p_{(4,3)} = \frac{F(3,3)}{F(4,3)} = \frac{1}{5}$. Then, the probability of color 3 being sampled is $p_{(3,3)} = \frac{a_3 \times F(2,2)}{F(3,3)} = 1$. Thus, one vertex with color 3 should be chosen and the probability of node 3 being sampled is $\frac{1}{2}$. Likewise, the probability of node 2 and 0 is $\frac{1}{2}$ and 1, respectively. Finally, the probability of $\{0, 3, 4\}$ being sampled is $\frac{1}{20}$.

B. Estimating the Number of k-Cliques

By Theorem 2, we can first make use of Algorithm 3 to uniformly sample k-color sets from G, and then estimate the clique density ρ_c in the k-color sets of G. After that, the number of k-cliques in G can be estimated by $\rho_c \times F(\chi, k)$. Based on this idea, we propose a weighted sampling algorithm to estimate the number of cliques in the dense regions of G. The detailed implementation of our algorithm is shown in Algorithm 4.

Let S be a set of nodes whose neighborhood subgraphs are *dense regions* of G, i.e., $\bar{d}(G(N_v(\vec{G}))) \geq k$ for each $v \in S$. Algorithm 4 first colors the graph using a linear-time greedy algorithm [32], [33] (line 1). Then, the algorithm invokes the DPCount procedure to compute the number of k-color sets for each $v \in S$ (lines 3-4). Let $cntKCol$ be the total number of k-color sets (line 5). Then, we can obtain a probability distribution D over S where $p(v) = F_v(\chi, k-1)/cntKCol$ for each $v \in S$ (line 6). After that, Algorithm 4 draws t k-color sets by (1) sampling a node $v \in S$ with probability $p(v)$ (line 9), and (2) uniformly sampling a $(k-1)$-color set from $G(N_v(\vec{G}))$ (line 10). The algorithm computes the k-clique density ρ_c in the sampled k-color sets (lines 11-13), and then estimates the k-clique count as $\rho_c \times cntKCol$ (line 14). The following theorem shows that Algorithm 4 can obtain an unbiased estimator.

Algorithm 4: Estimating the Number of k-Cliques by k-Color Set Sampling.

Input: A graph $\vec{G}$, a node set S, an integer k, and the sample size t
Output: An estimation of the number of k-cliques

1 Coloring the graph $\vec{G}$ using a linear-time algorithm [32], [33];
2 Let χ be the number of colors obtained;
3 **foreach** $v \in S$ **do**
4 $\quad F_v \leftarrow \mathsf{DPCount}(G(N_v(\vec{G})), \chi, k-1)$;
5 $cntKCol \leftarrow \sum_{v \in S} F_v(\chi, k-1)$;
6 Set the probability distribution D over the nodes in S where $p(v) = F_v(\chi, k-1)/cntKCol$ for each $v \in S$;
7 $successTimes \leftarrow 0$;
8 **for** $i = 1$ *to* t **do**
9 $\quad$ Independently sample a node v from D;
10 $\quad R \leftarrow \{v\} \cup \mathsf{DPSampler}(G(N_v(\vec{G})), \chi, k-1)$;
11 $\quad$ **if** R *is a* k*-clique* **then**
12 $\quad\quad successTimes \leftarrow successTimes + 1$;
13 $\rho_k \leftarrow successTimes/t$;
14 **return** $\rho_k \times cntKCol$;

Theorem 4: Algorithm 4 outputs an unbiased estimator for the number of k-cliques in the dense regions of G.

Proof: Let $X_i = 1$ if the i_{th} sampled k-color set is a k-clique, otherwise $X_i = 0$. Observe that

$$\Pr(X_i = 1) = \sum_{v \in S} [\Pr(choose\ v\ from\ D)$$

$$\times \Pr(choose\ a\ clique\ from\ G(N_v(\vec{G})))]. \quad (7)$$

In the summation, the former probability is $\frac{F_v(\chi,k-1)}{\sum_{v\in S} cnt_k(G(N_v(\vec{G})),color)}$, and the latter is exactly $\frac{cnt_k(G(N_v(\vec{G})),clique)}{F_v(\chi,k-1)}$. Consequently, we have $Pr(X_i=1)=\frac{\sum_{v\in S} cnt_k(G(N_v(\vec{G})),clique)}{\sum_{v\in S} cnt_k(G(N_v(\vec{G})),color)}$. This implies that the probability of sampling a k-clique is exactly the k-clique density in the dense regions. By the linearity of expectation, we have

$$\begin{aligned} &E\left[cntKCol \times \frac{\sum_{i\le t} X_i}{t}\right] \\ &= \sum_{v\in S} cnt_k(G(N_v(\vec{G})), color) \times \frac{\sum_{i\le t} E[X_i]}{t} \\ &= \sum_{v\in S} cnt_k(G(N_v(\vec{G})), clique). \end{aligned} \quad (8)$$

Therefore, Algorithm 4 returns an unbiased estimator of the k-clique count in the dense regions of G.□

By applying the classic Chernoff bound, we can easily derive that Algorithm 4 is able to produce a $1-\epsilon$ approximation of the k-clique count in the dense regions of the graph.

Theorem 5: Algorithm 4 returns a $1-\epsilon$ approximation of the number of k-cliques in the dense regions of G with probability $1-2\sigma$ if $t \ge \frac{3}{\rho_c\epsilon^2}\ln\frac{1}{\sigma}$, where ϵ and σ are small positive values and t is the sample size.

Proof: Denote by $\hat{\rho}_c$ the estimator of the k-clique density (line 13 of Algorithm 4). Since our estimator is unbiased, we have $E[\hat{\rho}_c]=\rho_c$. Then, the expected number of k-cliques in the t samples is $E[\hat{\rho}_c t]=\rho_c t$. Based on the Chernoff bound, we easily obtain the following results:

$$\Pr(\hat{\rho}_c t \le (1-\epsilon)\rho_c t) \le \exp\left(-\frac{\epsilon^2\rho_c t}{2}\right) \le \exp\left(-\frac{\epsilon^2\rho_c t}{3}\right), \quad (9)$$

$$\Pr(\hat{\rho}_c t \ge (1+\epsilon)\rho_c t) \le \exp\left(-\frac{\epsilon^2\rho_c t}{3}\right). \quad (10)$$

Further, we have:

$$\Pr\left(\frac{|\hat{\rho}_c-\rho_c|}{\rho_c} \ge \epsilon\right) \le 2\exp\left(-\frac{\epsilon^2\rho_c t}{3}\right). \quad (11)$$

Let $\exp(-\frac{\epsilon^2\rho_c t}{3}) \le \sigma$. Then, we can derive that $t \ge \frac{3}{\rho_c\epsilon^2}\ln\frac{1}{\sigma}$. This completes the proof.□

Note that by Theorem 5, the sample size of our algorithm relies on the k-clique density ρ_c. Since ρ_c is often not very small in the dense regions of a graph, Algorithm 4 is expected to be very efficient in practice which is also confirmed in our experiments. Below, we analyze the time and space complexity of Algorithm 4.

Theorem 6: Algorithm 4 consumes $O((|S|+t)\chi k + k^2 t + m + n)$ time and $O(m+n+\chi k)$ space.

Proof: For the time complexity, Algorithm 4 takes $O(m+n)$ time to obtain a feasible graph coloring. Then, it consumes $O(|S|\chi k)$ time to compute F_v for each $v \in S$. After that, to draw a k-color set, the algorithm takes $O(\chi k)$ time and $O(k^2)$ time to check whether it is a clique. Thus, the total time used in the k-color set sampling stage is $O(t(\chi k + k^2))$. As a consequence, the time complexity of Algorithm 4 is $O((|S|+t)\chi k + m + n + k^2 t)$. For the space complexity, the algorithm needs to store the graph G and the colors which takes $O(m+n)$ space in total. Additionally, the algorithm uses $O(\chi k)$ space to store the DP table when sampling a k-color set. Note that the algorithm does not store all the DP tables for all samples. Thus, the total space overhead of Algorithm 4 is $O(m+n+\chi k)$.□

Remark: The proposed k-color set sampling algorithm is completely different from the traditional color coding technique [19], [23], [26] for k-clique counting. The color coding technique randomly assigns a *color* to each node (it is actually not a valid graph coloring), in which two adjacent nodes may have the same color. However, our k-color set based sampling algorithm is based on the graph coloring technique which requires two adjacent nodes having different colors. For the color coding technique, the probability of each k-clique being colored with k different colors is $\frac{k!}{k^k}$ [19]. With the increase of k, such a probability decreases dramatically. However, our technique can ensure that the k-clique of G is a k-color set no matter what k is. Moreover, unlike color coding, the probability of sampling k nodes with k different colors from G (the colored graph) is nonuniform in our algorithm.

V. Connected k-Color Set Sampling

Recall that to achieve a $1-\epsilon$ approximation, the sample size of Algorithm 4 heavily relies on the k-clique density over the k-color sets, i.e., ρ_c (see Theorem 5). Although the dense regions of a graph G often have a relatively high ρ_c, it may still be very small in some cases as the k-color sets do not fully capture the clique property. To improve the effectiveness of the sampling algorithm, we propose a novel technique which can further boost the k-clique density by considering the connectivity of the k-color set.

A k-color set is definitely not a k-clique if the subgraph induced by the k-color set is not connected. Clearly, such disconnected k-color sets are unpromising samples for our sampling algorithm. Therefore, to improve the sampling performance, a natural question is that can we directly sample the connected k-color sets from G? In this section, we answer this question affirmatively by devising a novel k-color path sampling technique. The insight is that we only sample the k-color set in which there exists a simple path with length $k-1$ in the subgraph induced by the k-color set. For convenience, we refer to such a connected k-color set as a k-color path.

Similar to sampling k-color sets in G, we also need to uniformly sample the k-color paths. Unfortunately, the solutions proposed in Section IV are no longer applicable for sampling k-color paths. Below, we develop a new DP-based sampling technique to uniformly generate the k-color paths.

A. DP-Based k-Color Path Sampling

Counting the number of k-color paths: We start by developing an algorithm to count the number of k-color paths in a graph G. We assume that the graph G is colored with the color values

selected from $[1, \chi]$. Based on the color values, we can obtain a *color ordering* by sorting the nodes in a non-decreasing ordering of their color values. Note that we can use the nodes IDs to break ties to obtain a total ordering. It is worth mentioning that such a color ordering was used in the k-clique listing algorithms [16]. Clearly, we are able to construct a DAG $\vec{G}$ by the color ordering, where a directed edge $(u, v) \in \vec{G}$ is obtained by orienting the direction of $(u, v) \in G$ if v comes after u in the color ordering. Based on the DAG $\vec{G}$, we can obtain the following results.

Theorem 7: Let $\vec{G}$ be the DAG generated by the color ordering. Then, any $(k-1)$-path in $\vec{G}$ forms a k-color path.

Proof: Let $P = \{(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)\}$ be a $(k-1)$-path in $\vec{G}$. By the color ordering, we have $c(v_i) \leq c(v_{i+1})$ for every $i \in [1, k-1]$, where $c(v_i)$ denotes the color value of v_i. Since any two adjacent nodes have different colors, we have $c(v_i) \neq c(v_{i+1})$ for each $i \in [1, k-1]$. As a result, the path S is a k-color path.□

Theorem 8: Let $\vec{G}$ be the DAG generated by the color ordering. Then, any k-clique $C = \{v_1, v_2, \ldots, v_k\}$ in G is a k-color path in $\vec{G}$.

Proof: Let $C = \{v_1, v_2, \ldots, v_k\}$ be a k-clique in G. Clearly, the nodes in C have different colors. Suppose without loss of generality that $c(v_1) < c(v_2), \ldots, c(v_k)$. Since $\vec{G}$ is generated by the color ordering, there must exist a path $\{(v_1, v_2), \ldots, (v_{k-1}, v_k)\}$ in $\vec{G}$ which also forms a valid k-color path.□

Note that a k-color path in $\vec{G}$ does not necessarily form a k-clique in G. However, the set of k-color paths is clearly a subset of the set of k-color sets. Thus, the k-clique density over the k-color paths, denoted by ρ_p, must be no smaller than the k-clique density over the k-color sets.

Example 2: Reconsider the graph shown in Fig. 1(a). Clearly, we have $c(0) < c(1) = c(2) < c(3) = c(5) < c(4) = c(6)$. Fig. 1(c) plots all the 3-color paths, and Fig. 1(d) shows all the 4-color paths. The paths with dashed circles are not cliques, while the others are cliques. We can also easily derive that the 3-clique density is $\frac{6}{10}$ and the 4-clique density is $\frac{1}{3}$. As expected, the count of k-color paths is much smaller than the count of k-color sets.

To estimate the number of k-cliques in G, we need to compute ρ_p and the number of k-color paths as well. Let $\vec{G}_{v_i}$ be a subgraph of $\vec{G}$ induced by $\{v_i, \ldots, v_n\}$. Denote by $H(v_i, j)$ the number of j-paths containing the node v_i in $\vec{G}_{v_i}$. Clearly, each j-path containing v_i in $\vec{G}_{v_i}$ must start from v_i, since the node v_i in $\vec{G}_{v_i}$ only has out-neighbors. Thus, the total number of $(k-1)$-paths of $\vec{G}$, denoted by $\mathsf{cnt}_{k-1}(\vec{G}, path)$, can be computed by the following formula:

$$\mathsf{cnt}_{k-1}(\vec{G}, path) = \sum_{v_i \in \vec{G}} H(v_i, k-1). \tag{12}$$

Observe that the second node in each $(k-1)$-path containing v_i in $\vec{G}_{v_i}$ must be an out-neighbor of v_i. Thus, if we have the count of the $(j-2)$-paths containing v_x in $\vec{G}_{v_x}$ for each $v_x \in N_{v_i}(\vec{G}_{v_i})$, the count of $(j-1)$-paths containing v_i in G_{v_i} can be easily obtained. Specifically, we have the following recursive equation:

$$H(v_i, j) = \sum_{v_x \in N_{v_i}(\vec{G}_{v_i})} H(v_x, j-1). \tag{13}$$

Initially, we have

$$\begin{cases} H(v_i, 0) = 1, & \text{for all } i \in [1, n], \\ H(v_i, j) = 0, & \text{for all } i \in [1, n], j \in [1, k-1]. \end{cases} \tag{14}$$

Based on (12), (13) and (14), we can easily devise a DP algorithm to compute $\mathsf{cnt}_{k-1}(\vec{G}, path)$ which is detailed in the DPPathCount procedure of Algorithm 5 (lines 5-12). It is easy to derive that the time complexity of DPPathCount is $O(kn\chi)$, where χ is the maximum color value of G. This is because the cardinality of the out-neighbors for any node in $\vec{G}$ is bounded by $O(\chi)$.

Sampling a uniform k-color path: Similar to the DP-based sampling technique developed in Section IV-A, here we also propose a DP-based sampling algorithm to uniformly sample the k-color paths. Suppose without loss of generality that there is a randomly sampled k-color path of $\vec{G}$ starting from a node v, denoted by P_v. Then, for the second node in P_v, it must be an out-neighbor of v in $\vec{G}$. According to the DP equation ((13)), the number of $(k-1)$-paths starting from v is equal to the sum of the number of $(k-2)$-paths starting from each node in $N_v(\vec{G})$. Therefore, the next node of a random k-color path starting from v, denoted by u, should be drawn from $N_v(\vec{G})$ with probability $\frac{H(u,k-2)}{H(v,k-1)}$ by (13). We can recursively perform this sampling

Algorithm 5: DPPathSampler(G, k).

Input: A colored graph $G = (V, E)$, and an integer k
Output: A uniformly sampled k-color path

1 $\vec{G} \leftarrow$ the DAG generated by the color ordering of G;
2 $H \leftarrow$ DPPathCount$(\vec{G}, k)$;
3 $R \leftarrow$ DPPathSampling$(\vec{G}, H, k)$;
4 **return** R;
5 **Procedure** DPPathCount$(\vec{G}, k)$
6 $H(v_i, j) \leftarrow 0$, for $i \in [1, n]$ and $j \in [1, k-1]$;
7 **foreach** $i = 0$ *to* n **do** $H(v_i, 0) = 1$;
8 **foreach** $j = 1$ *to* $k-1$ **do**
9 **for** $i = 1$ *to* n **do**
10 **for** $v_x \in N_{v_i}(\vec{G})$ **do**
11 $H(v_i, j) \leftarrow H(v_i, j) + H(v_x, j-1)$;
12 **return** H;
13 **Procedure** DPPathSampling$(\vec{G}, H, k)$
14 $R \leftarrow \emptyset$; $Q \leftarrow V$;
15 **for** $i = 0$ *to* $k-1$ **do**
16 $\mathsf{cnt} \leftarrow \sum_{u \in Q} H(u, k-i-1)$;
17 Set the probability distribution D over the nodes in Q where $p(u) = H(u, k-i-1)/\mathsf{cnt}$ for each $u \in Q$;
18 Sample a node u from D;
19 $R \leftarrow R \cup \{u\}$; $Q \leftarrow N_u(\vec{G})$;
20 **return** R;

procedure to obtain a k-color path. The detailed implementation of this sampling technique is shown in Algorithm 5.

Algorithm 5 first constructs a DAG $\vec{G}$ by the color ordering (line 1). Then, the algorithm invokes DPPathCount to derive the DP table H (line 2). After that, Algorithm 5 calls the DPPathSampling procedure to uniformly sample a k-color path (line 3). Specifically, when sampling a node u from $N_v(\vec{G})$, DPPathSampling needs to set a probability distribution D over the set $N_v(\vec{G})$ based on (13) (lines 16-18). After choosing a node u, DPPathSampling turns to sample the next node from $N_u(\vec{G})$ (line 19). The DPPathSampling procedure terminate when k nodes are sampled.

It is important to note that Algorithm 5 can always obtain a k-color path if the DAG $\vec{G}$ contains at least one k-color path. This is because in lines 16-18, if a node u is sampled, then $H(u, k-i-1)$ must be larger than 0, indicating that the out-neighborhood $N_u(\vec{G})$ must be non-empty. As a consequence, if there is a k-color path in $\vec{G}$, the **for** loop in line 15 of Algorithm 5 will be executed k times which results in a k-color path. The following theorem shows that Algorithm 5 can obtain a uniform k-color path.

Theorem 9: Algorithm 5 outputs a uniform k-color path.

Proof: Consider a path $\{v_1, v_2, \ldots, v_k\}$. Let X be the event of this path being sampled by Algorithm 5. Denote by Y_i the event of a node v_i appearing in the path. Clearly, the probability of the first node v_1 being sampled is $\Pr(Y_1) = \frac{H(v_1,k-1)}{\sum_{u\in V} H(u,k-1)}$. Observe that in the i^{th}-iteration of the **for** loop (line 15), the distribution D for node v_i is constructed from $N_{v_{i-1}}(\vec{G})$. The node v_i being sampled in the **for** loop can be represented as an event $Y_i|Y_{i-1}$ (conditioned on Y_{i-1}), thus we have $\Pr(Y_i|Y_{i-1}) = \frac{H(v_i,k-i)}{\sum_{u\in N_{v_{i-1}}(\vec{G})} H(u,k-i)}$. As a consequence, we have

$$\begin{aligned}\Pr(X) &= \Pr(Y_1) \times \Pr(Y_2|Y_1) \times \cdots \times \Pr(Y_k|Y_{k-1}) \\ &= \frac{H(v_1,k-1)}{\sum_{u\in V} H(u,k-1)} \times \frac{H(v_2,k-2)}{\sum_{u\in N_{v_1}(\vec{G})} H(u,k-2)} \times \\ &\cdots \times \frac{H(v_k,0)}{\sum_{u\in N_{v_{k-1}}(\vec{G})} H(u,0)} = \frac{1}{\sum_{u\in V} H(u,k-1)}.\end{aligned} \tag{15}$$

Since the number of k-color paths in G is equal to $\sum_{u\in V} H(u,k-1)$, each k-color path is sampled uniformly.□

We analyze the time and space complexity of Algorithm 5 in the following theorem.

Theorem 10: Given an input graph G with n nodes and m edges, Algorithm 5 takes $O(\chi nk + m)$ time and uses $O(kn + m)$ space, where χ is the maximum color value.

Proof: First, the algorithm consumes $O(m + n)$ time to obtain a DAG. Second, as above analyzed, the DPPathCount procedure takes $O(nk\chi)$ time. Third, the DPPathSamplingprocedure uses $O(n + \chi k)$ time. This is because setting the probability distribution for the first node takes $O(n)$ time, while for the other nodes it takes at most $O(\chi)$ time. Thus, the total time complexity of Algorithm 5 is $O(\chi nk + m)$. For the space complexity, the algorithm needs to store the DAG and the DP table H which uses $O(nk + m)$ space in total.□

B. Estimating the k-Clique Counts

Based on Algorithm 5, we can devise a weighted sampling algorithm to construct an unbiased estimator to compute the number of k-cliques. Specifically, we can slightly modify Algorithm 4 by (1) replacing the DPCountprocedure in line 4 of Algorithm 4 with the DPPathCount procedure, and (2) replacing DPSampling in line 10 of Algorithm 4 with DPPathSampling. Due to the space limit, we omit the details of this modified algorithm. Similar to Theorems 4 and 5, the estimator based on the k-color path sampling is also unbiased, and the sample size can also be bounded by using the Chernoff bound. Moreover, it is easy to check that the sample size is no larger than that of Algorithm 4, because $\rho_p \geq \rho_c$.

For the time complexity, such a modified algorithm takes $O(|S|\delta^2 k)$ to compute the DP tables (i.e., H) for all nodes in S (because the input graph $G(N_v(\vec{G}))$ for the DPPathCount procedure has at most δ nodes), and consumes $O(\delta k + k^2)$ to sample a k-color path. Thus, the total time complexity of the algorithm is $O(|S|\delta^2 k + (\delta k + k^2)t + m + n)$, where $O(m + n)$ is taken for computing the graph coloring. The space overhead of the modified algorithm is $O(m + n + \delta k)$, because the DP table takes $O(\delta k)$ space.

VI. Colorful Triangle-Path Sampling

Note that k-color paths can significantly remove the unpromising k-color sets by introducing a connective constraint (i.e., a k-color set must form a path). However, the k-color path is still a very sparse structure, which does not fully capture the clique property. Specifically, k-color path only guarantees the existence of $k-1$ edges, which is the smallest number of edges to maintain the connectivity. In this section, we further develop a new technique, called k-triangle path, to prune those unpromising k-color paths that are not the k-cliques. In a simple path, any two consecutive nodes form a 2-clique. Similarly, we define the concept of *triangle-path*. In a triangle-path, any three consecutive vertices form a triangle. When the nodes of a triangle-path have distinct colors, the triangle-path is called a colorful triangle-path. In the following, we use k-triangle path to refer to a colorful triangle-path with k nodes.

The k-triangle path can capture the clique property better than k-color path, which can further improve the clique density. However, compared to k-color path, k-triangle path is a more complex structure. It is nontrivial to design efficient algorithms for uniformly sampling k-triangle paths. Below, we propose a new DP algorithm to achieve this goal.

A. DP-Based k-Triangle Sampling

As described in Section V-A, we assume that the graph G is colored with the color values selected from $[1, \chi]$. We can construct a DAG $\vec{G}$ based on the color ordering. Below, we formally define the concept of k-triangle path.

Definition 2: A k-triangle path is a k-color set with vertices $\{v_1, v_2, v_3, \ldots, v_k\}$ where $c(v_i) < c(v_{i+1})$ for all $i \in [1, k-1]$ and v_i, v_{i+1}, v_{i+2} form a triangle for all $i \in [1, k-2]$.

Let $\vec{G}$ be the DAG generated by the color ordering. Counting the colorful triangle-paths in G is equivalent to counting the triangle-paths in $\vec{G}$. Then, any k-clique $C = \{v_1, v_2, \ldots, v_k\}$ in G is a k-triangle path in $\vec{G}$. As described in Section V-A, the set of k-color paths is a subset of the set of k-color sets. Similarly, the set of k-triangle paths is a subset of k-color paths. Thus, the k-clique density over the k-triangle paths, denoted by ρ_t, must be no less than the k-clique density over the k-color paths.

Example 3: In Fig. 1(d), the two 4-color paths in the box are k-triangle paths. For example, in the 4-color path $\{0, 1, 5, 6\}$, both $\{0, 1, 5\}$ and $\{1, 5, 6\}$ are triangles. However, in the 4-color path $\{0, 2, 3, 6\}$, the three consecutive nodes $\{2, 3, 6\}$ do not form a triangle, thus $\{0, 2, 3, 6\}$ is not a k-triangle path.

Counting the number of k-triangle path: Denote by $T((v_x, v_y), j)$ the number of j-triangle-paths with (v_x, v_y) as the first edge. Then, the total number of k-color paths of $\vec{G}$, denoted by $\mathsf{cnt}_k(G, triangle)$, can be computed by the following formula:

$$\mathsf{cnt}_k(G, triangle) = \sum_{(v_i, v_j) \in \vec{G}} T((v_i, v_j), k). \tag{16}$$

Let $\vec{G}_{v_i}$ be a subgraph of $\vec{G}$ induced by $\{v_i, \ldots, v_n\}$. Since the node v_x in $\vec{G}_{v_x}$ only has out-neighbors, other nodes in the k-triangle paths are in $\vec{G}_{v_x}$. Denote by v_z the third node in a j-triangle-path. It is easy to see that v_z is the common neighbor of v_x and v_y, because the three consecutive nodes in a k-triangle paths must form a triangle. Based on this property, we can derive the following equation:

$$T((v_x, v_y), j) = \sum_{v_z \in N_{v_x}(\vec{G}_{v_x}) \cap N_{v_y}(\vec{G}_{v_y})} T((v_y, v_z), j-1). \tag{17}$$

Initially, we have

$$T((v_x, v_y), 2) = 1, \forall (v_x, v_y) \in E. \tag{18}$$

Based on these equations, we can easily devise a DP algorithm to compute $\mathsf{cnt}_k(G, triangle)$. The detailed implementation of this DP algorithm is shown in the TriPathCount procedure of Algorithm 6 (lines 5-11). Specifically, Line 6 initializes the DP table based on (18), and lines 7-10 is the DP procedure based on (17).

Sampling a uniform k-triangle path: Similar to the algorithm to uniformly sample the k-color paths, we propose a DP-based sampling algorithm to uniformly draw k-triangle paths. Suppose that there is a randomly selected k-triangle path, denoted by P. With (16), we can derive that the probability of P starting by edge (v_x, v_y) is $\frac{T((v_x, v_y), k)}{\mathsf{cnt}_k(G, triangle)}$. Then for the third node v_z in P, it must be the common out-neighbor of v_x and v_y. According to (17), the number of k-triangle paths with (v_x, v_y) as the first edge is equal to the sum of $(k-1)$-triangle paths with (v_y, v_z) as the first edge, thus the probability of v_z being sampled is $\frac{T((v_y, v_z), k-1)}{T((v_x, v_y), k-2)}$. Similar mechanism can be applied to sample the next nodes. The detailed implementation is shown in Algorithm 6.

Algorithm 6 first constructs a DAG $\vec{G}$ by the color ordering (line 1). Then, the algorithm invokes TriPathCount to derive the DP table T and calls the DPTriSampling procedure to uniformly sample a k-color path (line 3). Based on (17), DPTriSampling sets a probability distribution D over the set of edges (line 13) and samples the first two nodes v_x and v_y according to D (lines 14-15). With the first two nodes, the set of the third nodes is the common out-neighbors of v_x and v_y (line 15). Then, DPTriSampling samples the next node from Q by setting a probability distribution over Q (lines 17-20). The DPTriSampling procedure terminates when k nodes are sampled.

A k-triangle path can always be obtained by Algorithm 6 if the DAG $\vec{G}$ contains at least one k-triangle path. This is because in lines 17-20, if a node v_z is sampled, then $T((v_y, v_z), k-i+2)$ must be larger than 0, indicating that the common out-neighbor of v_y and v_z must be non-empty (line 21). As a consequence, the **for** loop in line 16 of Algorithm 6 will be executed $k-2$ times which results in a k-triangle path. The following theorem shows that Algorithm 6 can obtain a uniform k-color path.

Theorem 11: Algorithm 6 outputs a uniform k-triangle path.

Proof: Consider a path $\{v_1, v_2, \ldots, v_k\}$. Let X be the event of this path being sampled by Algorithm 6. Denote by Y_i the event of two nodes v_{i-1} and v_i appearing in the

Algorithm 6: DPTriSampler(G, k).

Input: A colored graph $G = (V, E)$, and an integer k
Output: A uniformly sampled k-triangle path

1 $\vec{G} \leftarrow$ the DAG generated by the color ordering of G;
2 $T \leftarrow \mathsf{TriPathCount}(\vec{G}, k)$;
3 $R \leftarrow \mathsf{DPTriSampling}(\vec{G}, T, k)$;
4 **return** R;
5 **Procedure** $\mathsf{TriPathCount}(\vec{G}, k)$
6 $\quad T((v_x, v_y), 2) \leftarrow 1$, for $(v_x, v_y) \in E$;
7 $\quad$ **foreach** $j = 3$ *to* k **do**
8 $\quad\quad$ **for** $(v_x, v_y) \in E$ **do**
9 $\quad\quad\quad$ **for** $v_z \in N_{v_x}(\vec{G}_{v_x}) \cap N_{v_y}(\vec{G}_{v_y})$ **do**
10 $\quad\quad\quad\quad T((v_x, v_y), j) \leftarrow T((v_x, v_y), j) + T((v_y, v_z), j-1)$;
11 $\quad$ **return** T;
12 **Procedure** $\mathsf{DPTriSampling}(\vec{G}, T, k)$
13 $\quad$ Set the probability distribution D over the edges in E where $p(e) = T(e, k) / \sum_{e \in E} T(e, k)$;
14 $\quad$ Sample an edge (v_x, v_y) from D;
15 $\quad R \leftarrow \{v_x, v_y\}$; $Q \leftarrow N_{v_x}(\vec{G}_{v_x}) \cap N_{v_y}(\vec{G}_{v_y})$;
16 $\quad$ **for** $i = 3$ *to* k **do**
17 $\quad\quad \mathsf{cnt} \leftarrow \sum_{v_z \in Q} T((v_y, v_z), k-i+2)$;
18 $\quad\quad$ Set the probability distribution D over the nodes in Q where $p(v_z) = T((v_y, v_z), k-i+2)/\mathsf{cnt}$ for each $v_z \in Q$;
19 $\quad\quad$ Sample a node v_z from D;
20 $\quad\quad R \leftarrow R \cup \{v_z\}$;
21 $\quad\quad Q \leftarrow N_{v_y}(\vec{G}_{v_y}) \cap N_{v_z}(\vec{G}_{v_z})$;
22 $\quad\quad v_y \leftarrow v_z$;
23 $\quad$ **return** R;

path. Clearly, the probability of the first two nodes v_1 and v_2 being sampled is $\Pr(Y_1) = \frac{T((v_1,v_2),k)}{\sum_{e\in E} T(e,k)}$. Observe that in the i^{th}-iteration of the **for** loop (line 16), the nodes in Q are the common out-neighbor of v_{i-1} and v_{i-2}. Thus, the event that a node v_i is sampled in the i^{th} **for** loop can be represented as $Y_i|Y_{i-1}$ (conditioned on Y_{i-1}). We have $\Pr(Y_i|Y_{i-1}) = \frac{T((v_{i-1},v_i),k-i+2)}{\sum_{u\in N_{v_{i-1}}(\vec{G}_{v_{i-1}})\cap N_{v_{i-2}}(\vec{G}_{v_{i-2}})} T((v_{i-1},u),k-i+2)} = \frac{T((v_{i-1},v_i),k-i+2)}{T((v_{i-2},v_{i-1}),k-i+3)}$. As a result, we have

$$\begin{aligned}\Pr(X) &= \Pr(Y_2)\times\Pr(Y_3|Y_2)\times\cdots\times\Pr(Y_k|Y_{k-1})\\ &= \frac{T((v_1,v_2),k)}{\sum_{e\in E}T(e,k)}\times\frac{T((v_2,v_3),k-1)}{T((v_1,v_2),k)}\times\\ &\quad\cdots\times\frac{T((v_{k-1},v_k),2)}{T((v_{k-2},v_{k-1}),3)}\\ &= \frac{1}{\sum_{e\in E}T(e,k)}.\end{aligned} \tag{19}$$

Since the number of k-triangle paths in G is equal to $\sum_{e\in E} T(e,k)$, each k-triangle path is sampled uniformly.□

Example 4: Let $k = 4$. In Fig. 1(a), there are two k-triangle paths $\{0,1,5,6\}$ and $\{0,2,3,4\}$. We have $T((0,1),4) = 1$, which means the count of k-triangle paths with head (0,1) is 1. So the probability of (0,1) being sampled as the first two vertices is $\frac{T((0,1),4)}{2} = \frac{1}{2}$ (line 13 of Algorithm 6). The set of common neighbors of 0 and 1 is $\{5\}$. Thus the probability of 5 being sampled as the third vertex is 1. Similarly, the set of common neighbors of 1 and 5 is $\{6\}$, and the probability of 6 being sampled is 1. At last, the probability of $\{0,1,5,6\}$ being sampled is $\frac{1}{2}\times 1\times 1 = \frac{1}{2}$. The probability of $\{0,2,3,4\}$ is also $\frac{1}{2}$. Therefore, the probability of each k-triangle path being sampled is equal.

Remark: Note that for all our algorithms, the clique density is a fixed value of a network. For example, for the k-triangle path sampling algorithm, the clique density is determined by the count of the cliques among the k-triangle paths. Reconsider the graph in Fig. 1(a), there is a 4-clique $\{0,2,3,4\}$ in the two triangle paths $\{0,2,3,4\}, \{0,1,5,6\}$, and the clique density is 0.5.

We analyze the time and space complexity of Algorithm 6 in the following theorem.

Theorem 12: The procedure TriPathCount in Algorithm 6 takes $O(k\triangle)$ time and uses $O(km)$ space, where $\triangle$ is the number of triangles of the input graph. The procedure DP-TriSampling in Algorithm 6 takes $O(m+\chi k)$ time.

Proof: It is easy to see that the total time costs of Line 8 and Line 9 in Algorithm 6 is bounded by $O(\triangle)$. Thus, the Tri-PathCount procedure takes at most $O(k\triangle)$ time. For the space complexity, the algorithm needs to store the DP table T which uses $O(mk)$ space. In DPTriSampling, setting the probability distribution for the first two nodes takes $O(m)$ time (line 13), while for the other nodes it takes at most $O(\chi)$ time. Thus, the total time complexity of DPTriSampling is $O(m+\chi k)$.□

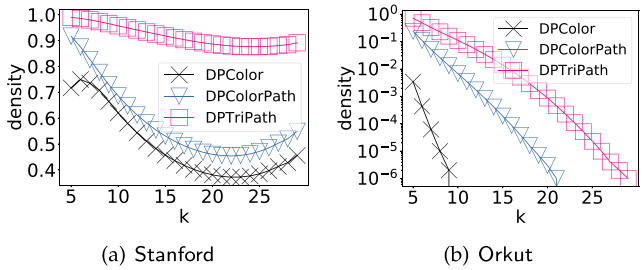


Fig. 2. Comparing ρ_c, ρ_p, and ρ_t for different k.

B. Estimating the k-Clique Counts

Similar to Sections IV-B and V-B, we can construct an unbiased k-clique estimator based on Algorithm 4 and Algorithm 6. Since the estimator is very similar to those shown in Sections IV-B and V-B, we omit the details for brevity.

For the time complexity, such a modified algorithm takes $O(\triangle_S k)$ to compute the DP tables (i.e., the DP table T in Algorithm 6) for all nodes in S where $\triangle_S$ is the sum of the number of triangle in the dense region S. It also consumes $O(\chi k + k^2)$ to sample only one k-triangle path, where $O(k^2)$ is the time to check whether the sampled k nodes is a k-clique. Thus, the total time complexity of the algorithm is $O(\triangle_S k + (\chi k + k^2)t + m + n)$, where $O(m+n)$ is taken for computing the graph coloring. The space overhead of the modified algorithm is $O(\delta^2 k)$, because the DP table takes $O(m'k)$ space where m' is the maximum number of edges for $\vec{G}_v$ and it must satisfy $m' \le \delta^2$.

C. Discussion

In this subsection, we analyze the relationships among three proposed algorithms and analyze in which case k-triangle paths is better than k-color sets and k-color paths.

According to Theorem 5, the sample size needed to compute an accurate estimate is $\frac{3}{\rho\epsilon^2}\ln\frac{1}{\sigma}$ where ρ is the clique density and ϵ, σ are small constant numbers. This bound is only determined by the clique density. In other words, if the sample size is fixed, the clique density is the only factor that has effect on the accuracy. When ρ is very small, it needs quite a large size of samples to achieve an accurate answer. All the Bernoulli-style sampling algorithms have this property [34].

Denote by ρ_c, ρ_p, ρ_t the clique density over the k-color sets, k-color paths, k-triangle paths, respectively. ρ_t is always the largest one as described in the following. Since a k-triangle path must be a k-color path and k-color path must be a k-color set, it is easy to derive that the set of k-triangle paths is a subset of k-color paths and the set of k-color paths is a subset of k-color sets. Thus it has $\rho_t \ge \rho_p \ge \rho_c$. For example, in Fig. 1(a), when $k = 4$, it has $\rho_c = \frac{1}{8}, \rho_p = \frac{1}{3}, \rho_t = \frac{1}{2}$ (as shown in Fig. 1(b) and (d)). Table II in experiment further shows the relationship. We also plot the change tendency of ρ_c, ρ_p and ρ_t on two representative datasets, Stanford and Orkut, in Fig. 2. In Fig. 2, ρ_t is the most largest and robust when k becomes large. Thus the estimator based on the k-triangle paths needs smaller sample size.

Algorithm 7: The Adaptive Sampling Framework.

Input: A graph $G = (V, E)$, the dense part S of G, an integer k, and the error bound ϵ
Output: A $(1-\epsilon)$-approximation of the density of k-cliques.

1 $C \leftarrow 0; T \leftarrow 0$;
2 $t \leftarrow 10000$;
3 $threshold \leftarrow \frac{3}{\epsilon^2}\ln\frac{1}{\sigma}$;
4 **while** $C < threshold$ **do**
5 Sampling$(\vec{G}, S, k, t)$;
6 $T \leftarrow T + t$;
7 $c \leftarrow$ *count of sampled cliques among the t samples*;
8 $C \leftarrow C + c$;
9 **if** $C > 0$ **then** $t \leftarrow threshold/C \times T$;
10 **else** $t \leftarrow t \times 10$;
11 **return** $\frac{C}{T}$;

However, smaller sample size does not mean less running time. According to Theorem 6, the proposed sampling based algorithms are composed of two steps. The first step is the computation of dynamic programming table. The second step is sampling t samples according to the distribution defined by the dynamic programming table. Since k-triangle path is a more complex structure than DPColor and DPColorPath, the computation of dynamic programming needs more running time, as described in Theorem 3, 10 and 12. Thus k-triangle path is better than k-color set and k-color path when ρ_p and ρ_c is quite smaller than ρ_t. As shown in Fig. 2, this happens when k becomes large on real-world networks.

VII. Adaptively Determining the Sample Size

In Algorithm 2, it needs to set the sample size t as a fixed value. The advantage of a fixed sample size is that the running time can be controlled by the parameter t. However, there is no confirmation that the results given by Algorithm 2 are accurate. To overcome this problem, we provide a new framework that can guarantee the accuracy.

The key idea of the new framework is based on the concept that an estimate is accurate if the number of cliques in the samples exceeds a threshold. We set the threshold as $\frac{3}{\epsilon^2}\ln\frac{1}{\sigma}$ according to Theorem 13. Theorem 13 explains the idea more clearly.

Theorem 13: Suppose that the sample size is t and the number of k-cliques in the t sample size is c. $\hat{\rho} = \frac{c}{t}$ is a $1-\epsilon$ approximation of ρ with probability $1 - 2\sigma$ if $c \geq \frac{3}{\epsilon^2}\ln\frac{1}{\sigma}$.

Proof: Since $c = \hat{\rho}t$, it has $t \geq \frac{3}{\hat{\rho}\epsilon^2}\ln\frac{1}{\sigma}$. Then the theorem can be proved by Theorem 5. □

Theorem 13 describes that $\hat{\rho}$ is accurate only if c is large enough, regardless of the value of t. Based on this idea, we design a new framework that keeps sampling until c is larger than the threshold. In the new framework, we utilize the *Adaptive Sampling* to adapt the sample size according to the existing sampling results. If there are C cliques in T samples already and we needs $threshold$ cliques in total, the following sample size should be $threshold/C \times T$.

The details of the new framework is shown in Algorithm 7. Algorithm 7 inputs an error bound ϵ and returns a $(1-\epsilon)$-approximation. At first, it samples 10^3 samples to test the clique density (line 2). If no clique is sampled, use more samples to test the clique density (line 10). If there exists cliques in the T samples, adjust the count of samples accordingly (line 9). The adjusting method in line 9 is a simple yet effective way to make the value of C approaching the threshold. At last, the approximation is returned (line 11).

Instead of time complexity, we analyze the upper bound of the sampling times of Algorithm 7, i.e. the value of T in Algorithm 7, which is the key to the running time. We omit the proof because it is quite clear.

Theorem 14: The sampling times of Algorithm 7 is $O(\max(10^4, \frac{3}{\rho\epsilon^2}\ln\frac{1}{\sigma}))$.

The advantage of the new framework is that it can guarantee the accuracy of the results. The disadvantage is that the time complexity of our algorithm depends on the clique density. Therefore, when k is large (e.g., $k > 25$), the clique density might be extremely small, resulting in that the algorithm requires a large number of samples to achieve a good accuracy guarantee. In this case, the algorithm may be costly to obtain a good approximation. Fortunately, for real-world applications, k is often not very large (e.g., $k < 20$), our algorithm is very efficient and extremely fast in practice as shown in our experiments. In fact, in subgraph counting field, there are no existing algorithms that have both polynomial time complexity and strong accuracy guarantee [35].

Example 5: To aid understanding, we describe how the adaptive sampling framework works on the Orkut network with $\epsilon = 0.05, \delta = 0.01$ and DPPathSampler. The $threshold$ in line 3 is 5519. The real clique density is 0.0132. At first, the framework samples 10^4 times and get 91 cliques. Now the estimated density is 0.0091 and the error is $\frac{0.0132-0.0091}{0.0132} = 0.31$, which is larger than ϵ. According to the adaptive sampling method, to let the count of the sampled cliques larger than $threshold$, we need $threshold/C \times T = 606483$ more samples (line 9). After sampling, there are 7837 cliques in the 606483 samples. Now there are $C = 7837 + 91$ cliques among the $T = 606483 + 10000$ samples, and the estimated clique density is 0.0129. The error is $\frac{0.0132-0.0129}{0.0132} = 0.02$, which is smaller than ϵ.

VIII. Experiments

A. Experimental Setup

We compare the proposed algorithms with three state-of-the-art k-clique counting algorithms which are kClist [15], [16], PIVOTER [17], TuranShadow [20]. The kClist algorithm is an exact k-clique counting algorithm which is based on k-clique enumeration [15]. Note that the original kClist algorithm is based on the degeneracy ordering. Li et al. [16] proposed an improved version based on a hybrid of the degeneracy and color ordering. In our experiment, kClist denotes such an improved version. PIVOTER and TuranShadow are the state-of-the-art exact and approximate k-clique counting algorithms respectively. Both PIVOTER and TuranShadow were proposed by Jain and

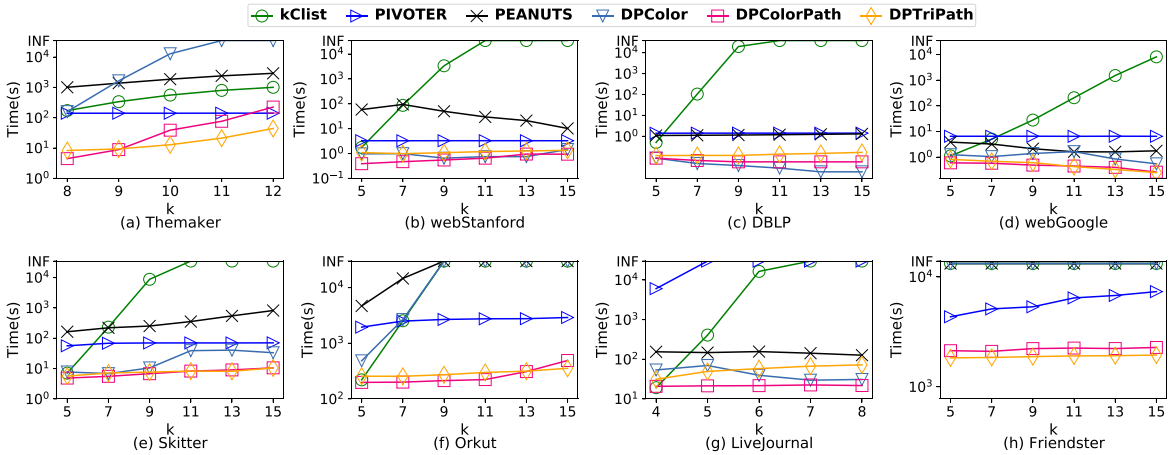


Fig. 3. Running time of different algorithms (the relative errors for PEANUTS, DPColor, DPColorPath and DPTriPath are set to 0.1%).

TABLE I
DATASETS

Networks	n	m	δ
Themaker	69, 413	3,289,686	164
Stanford	281,903	1,992,636	71
DBLP	425,957	1,049,866	113
Google	916,428	4,322,051	44
Skitter	1,696,415	11,095,298	111
Orkut	3,072,627	117,185,083	253
LiveJournal	4,036,538	34,681,189	360
Friendster	65,608,366	1,806,067,135	304

TABLE II
k-CLIQUE DENSITIES ($\rho_c/\rho_p/\rho_t$) IN THE DENSE REGIONS (%)

Networks	$k=8$	$k=15$
Themaker	0.001/0.003/0.02	0.0/0.0/1e-7
Stanford	70.8/77.7/97.3	45.8/53.1/91.7
DBLP	100.0/100.0/100.0	100.0/100.0/100.0
Google	91.2/94.4/96.0	84.7/85.7/85.9
Skitter	5.3/26.4/47.8	0.1/2.3/5.9
Orkut	0.0/2.6/22.8	0.0/0.0002/1.4
LiveJournal	80.4/91.0/95.0	-/-/-
Friendster	0.0/18.1/ 62.9	0.0/52.2/78.0

Seshadhri [17], [20]. PEANUTS [27] is an improved version of TuranShadow which is more efficient than TuranShadow, thus we use PEANUTS as the baseline instead of TuranShadow. The C++ codes of all these algorithms are publicly available, thus we use their implementations in our experiments. For our algorithms, we implement DPColor, DPColorPathand DPTriPath. The three algorithm are Algorithm 2 integrated with three sampling algorithms. All of them are implemented in C++. All algorithms are evaluated on a PC with two 2.1 GHz Xeon CPUs (16 cores in total) and 128 GB memory running CentOS 7.6.

Datasets: We use 8 large real-life datasets in our experiments. Table I summarizes the detailed statistic information of all datasets. The last column of Table I denotes the degeneracy of the graph. Stanfordand Google are web networks. DBLP is a co-authorship network, and Skitter is an internet graph. Themaker, Orkut, LiveJournal, and Friendster are social networks. All datasets are downloaded from (snap.stanford.edu) and (https://networkrepository.com/networks.php).

B. Experimental Results

Exp 1. Runtime of different algorithms: In this experiment, we compare the running time of different algorithms on all datasets. Note that for each approximation algorithm (PEANUTS, DPColor, DPColorPathand DPTriPath), we record its running time when the algorithm achieves a 0.1% relative error. Here the relative error is computed by $|f-\hat{f}|/f$, in which f is the exact k-clique count and $\hat{f}$ is the estimated count. For all algorithms, if they cannot terminate within 5 hours, we set their running time to "INF". Fig. 3 shows the running time of various algorithms.

We first compare our algorithms with kClist and PIVOTER. As can be seen, all of our algorithms DPColor, DPColorPath and DPTriPath are significantly faster than kClist and PIVOTER on most datasets with varying k. The kClist algorithm is generally intractable for large k on all datasets. On most datasets, DPColorPath is around one order of magnitude faster than PIVOTER. The hardest instance is the LiveJournal graph, on which PIVOTER only obtains the number of 4-cliques within 5 hours, whereas DPColorPath takes around 20 seconds to achieve a 0.1% relative error (DPColorPath can achieve at least three orders of magnitude faster than PIVOTER on LiveJournal). Note that since both kClist and PIVOTER are intractable on LiveJournal when $k \geq 6$, we use the exact k-clique count obtained from [36], where $k \leq 8$, to compute the relative errors for the approximation algorithms. Moreover, as reported in [36], the running time of such a GPU-parallelized PIVOTER algorithm using 5120 CUDA Cores is 6,851 seconds for $k=8$, while our *sequential* DPColorPath(DPColor) take around 20 seconds to obtain a very accurate k-clique count. On Orkut and LiveJournal, the exact algorithm PIVOTER is faster than DPColor. This is because the clique density over DPColor is almost zero in Orkut and Friendster, as shown in Table II. According to Theorem 5, it needs a large number of samples to guarantee the accuracy when

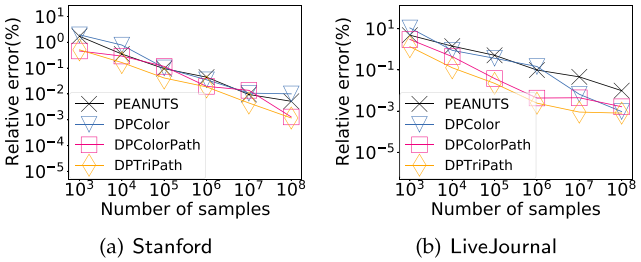


Fig. 4. Relative errors with varying sample size ($k = 8$).

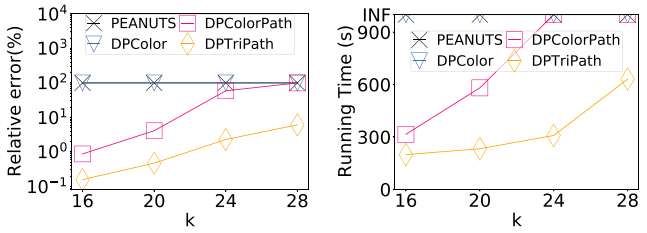


Fig. 5. Performance on Orkut when k is large.

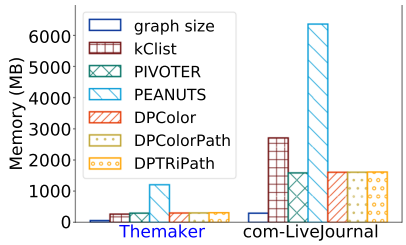


Fig. 6. Memory usage of various algorithms ($k = 8$).

the clique density is small. The results happens on the datasets that have very small cliques density. These experiment results indicate that our algorithms are extremely efficient for k-clique counting.

By comparing our algorithms with PEANUTS, we can see that DPColor, DPColorPath and DPTriPath are all consistently faster than PEANUTS on all datasets with varying k. On most datasets, DPColorPath is orders of magnitude faster than PEANUTS. For example, on DBLP, DPColor, DPColorPath and DPTriPath all take around 0.1 s, while PEANUTS consumes more than 1 seconds for most k values. In addition, on Orkut and Friendster, PEANUTS and DPColor cannot achieve a desired relative error within 5 hours for large k values, while DPColorPath and DPTriPath are still very efficient on these two datasets. For our algorithms, both DPColorPath and DPTriPath are generally faster than DPColor on large graphs. Moreover, the performance of DPColorPathand DPTriPathis much more stable than DP-Color on all datasets. Additionally, we can also see that in the graph Themaker and Friendster, DPTriPath is faster than DPColorPath. The reason is that DPTriPath can achieve a much higher clique density than DPColorPath, thus it needs much less samples to achieve 0.1% relative error. Although the complexity of DPTriPath is higher than DPColorPath to draw a sample, it needs much less samples, thus it can be faster than DPColorPath. These results confirm our theoretic analysis in Sections IV, V and VI.

Exp 2. Relative errors with varying sample size: Fig. 4 shows the relative errors of three algorithms with varying sample size on Stanford and LiveJournal. Similar results can also be observed on the other datasets. As shown in Fig. 4, the relative error of DPColorPath is lower than those of DPColor and PEANUTSon most cases, and DPTriPath is further lower than DPColorPath. When the sample size is 10^8, the relative error of DPColorPath is slightly larger than those of DPColoron LiveJournal. This is because the sample size is large enough to let the relative error of both the DPColorPath and DPColor be around 10^{-5}. In general, the relative errors of all algorithms decrease with the sample size increases. Moreover, we can see that both DPColorPath and DPTriPath obtain a 10^{-5} relative error on all datasets when the sample size is 10^8, indicating that DPColorPath and DPTriPath can achieve very high accuracy using a reasonable number of samples. These results further confirm the efficiency and effectiveness of our techniques.

Exp 3. Performance of different algorithms for a large k: In this experiment, we evaluate the performance of different sampling algorithms for a large k. We set the sample size as 5×10^7 on Orkut for all algorithms in Fig. 5(a). As can be seen, the error rates of all algorithms increase as k increases. DPColor and PEANUTS cannot obtain accurate and valid results for large k. Only DPTriPath can constantly achieve a relative error below 10% for large k. And DPTriPath consistently outperforms DPColorPath in at least one order of magnitude. This is because DPTriPath is more powerful to capture the clique property than DPColorPath, and the clique density over DPTriPath is larger than DPColorPath. Fig. 5(b) compares the running time of the algorithms to make the relative error below 10%. DPTriPath is more robust than DPColorPath when k becomes large. These results show the advantage of DPTriPath.

Exp 4. K-clique density: In this experiment, we evaluate the k-clique densities over the k-color sets (ρ_c), the k-color paths (ρ_p) and the k-triangle paths (ρ_t) in the dense regions of the graph, respectively. The results on all datasets are reported in Table II. As expected, ρ_c is lower than ρ_p, and ρ_p is lower than ρ_t on all datasets. Moreover, all ρ_c, ρ_p and ρ_t can achieve a very high value on most datasets. For example, on DBLP, all of them are near to 100%. In general, they decrease with k increases. Nevertheless, on most datasets, ρ_t is always very large even when $k = 15$. These results further confirm that the proposed techniques can achieve high accuracy on real-life graphs. Note that in Fig. 3(f) and (h), the exact algorithm PIVOTER is faster than the proposed DPColor algorithm. This is because the clique density over k-color set is almost zero on Orkut and Friendster, as shown in Table II. According to Theorem 5, it needs a large number of samples to guarantee the 0.1% accuracy.

Exp 5. Memory overheads: Fig. 6 shows the memory usages of various algorithms on Themaker and LiveJournal for $k = 8$. The results for the other k values and datasets are consistent. As expected, the space consumption of PEANUTS is significantly

TABLE III
RUNTIME OF OUR PARALLEL ALGORITHMS ($k = 8$, $t = 5 \times 10^6$, SEC.)

Datasets	Algorithms	Threads				
		1	4	8	12	16
LiveJournal	DPColor	24.8	7.1	4.3	2.7	2.1
	DPColorPath	28.4	7.5	3.9	2.7	2.1
	DPTriPath	142.67	37.98	18.94	12.80	9.81
Friendster	DPColor	2481.5	650.3	341.6	244.4	196.5
	DPColorPath	2132.2	559.6	293.3	210.3	171.9
	DPTriPath	2430.32	636.94	336.31	239.68	197.01

TABLE IV
RATIO OF THE k-CLIQUES IN THE SPARSE REGIONS

Networks	$k = 3$	$k = 8$	$k = 12$	$k = 15$
Themaker	1.53%	1.24%	0.02%	0.30%
Stanford	6.15%	0.01%	0.00%	0.00%
DBLP	33.11%	0.00%	0.00%	0.00%
Google	11.63%	6.47%	0.69%	0.36%
Skitter	19.57%	0.03%	0.00%	0.00%
Orkut	6.47%	0.07%	0.00%	0.00%
LiveJournal	9.30%	0.00%	0.00%	0.00%
Friendster	26.15%	0.30%	0.01%	0.00%

higher than the other algorithms, as it needs to store the partial Tuŕan Shadow structure. The space overheads of our algorithms and PIVOTER are comparable, while kClist consumes slightly more space than our algorithms. These results demonstrate that our algorithms are space efficient.

Exp 6. Parallel performance of our algorithms: In this experiment, we evaluate the parallel performance of our algorithms. To this end, we implement the parallel versions for DPColor, DPColorPath and DPTriPath using OpenMP. We fix the sample size as 5×10^6 to evaluate the runtime on the two largest datasets. The results are shown in Table III. As can be seen, all of DPColor, DPColorPath and DPTriPath can achieve $12\times \sim 14\times$ speedups when using 16 threads. This result indicates a high degree of parallelism of our algorithms.

Exp 7. The number of k-cliques in the sparse regions: In this experiment, we evaluate the number of k-cliques in the sparse regions of a graph on all datasets. Note that a node's neighborhood-induced subgraph is called a *sparse region* of a graph if the average degree of such a subgraph is smaller than k. Clearly, if the sparse regions have less number of k-cliques, the PIVOTER algorithm should be more efficient. Table IV reports our results on all datasets. As can be seen, for a relatively large k, the number of k-cliques in the sparse regions of all datasets only accounts for a small portion of the total number of k-cliques. On most datasets, such a ratio usually does not exceed 0.1%. These results indicate that the proposed framework, which integrates both PIVOTER and sampling techniques, can be very efficient for handling real-life graphs.

Exp 8. Maximum clique size in the sparse regions: Table V shows the maximum clique size, the maximum degeneracy among the subgraphs, i.e. $\max \delta_v$ where δ_v is the degeneracy of the subgraph $G(N_v(\vec{G}))$ and the value of $\lceil\sqrt{k\alpha + \frac{1}{2}}\rceil$ on the sparse regions of the graph (Theorem 1). Recall that the PIVOTER algorithm is based on the enumeration of maximal

TABLE V
MAXIMUM CLIQUE SIZE/ $\max \delta_v$ / $\lceil\sqrt{k\alpha + \frac{1}{2}}\rceil$ OF THE SPARSE REGIONS OF DIFFERENT GRAPHS

Networks	δ	$k = 3$	$k = 8$	$k = 12$	$k = 15$
DBLP	113	4/7/19	13/16/31	20/23/37	27/30/42
Skitter	111	8/11/19	20/23/30	27/30/37	35/38/41
Orkut	253	10/13/28	22/26/45	30/34/56	35/38/62
LiveJournal	360	8/11/33	19/22/54	24/28/66	28/31/74
Friendster	304	21/24/31	47/50/50	50/63/61	60/63/68

cliques, thus the maximum clique size bounds the recursion depth of PIVOTER. From Table V, we can observe that the maximum clique in the sparse region of each graph is relatively small compared with the degeneracy δ of the entire graph, where the degeneracy value is the upper bound of the maximum clique size. Moreover, the $\max \delta_v$ is much smaller than δ and $\lceil\sqrt{k\alpha + \frac{1}{2}}\rceil$ is also not very large, which further indicates the high effectiveness of the proposed solution.

Exp 9. Test different values of threshold to split network: Table VI shows the performance of DPColor, DPColorPath and DPTriPath on different values of threshold to split the networks. In Table VI, the total running time increases and the relative error decreases on most cases as the threshold increases. For example, on Orkut, the total running time of DPTriPath is $239.8\,s, 251.5\,s, 310.6\,s$ and the relative error is $0.25\%, 0.20\%, 0.20\%$ for the thresholds of $0.5\,k, k, 2\,k$ respectively. However, the results under different values of threshold differs no more than an order of magnitude. According to these results, we can conclude that our algorithm is not very sensitive to the threshold value.

Exp 10. Results with adaptive sample size: Table VII shows the performance of Algorithm 7 over different density. The value of δ is set as 0.01. In Table VII, the columns are (1) clique density, (2) the sampler, the value of k and the network, (3) the value of error bound ϵ in Algorithm 7, (4) the total sample size, i.e. the value of T in Algorithm 7, and (5) the estimate error. As shown in Table VII, no matter what the value of density, the estimate error is consistently smaller than the given expected error bound ϵ. The value of T tends to becomes larger when the clique density and the error bound ϵ becomes smaller. These results are consistent with Theorem 13, which confirms that Algorithm 7 can achieve a good accuracy guarantee.

Table VIII shows the running time of DPColor, DPColorPath and DPTriPath equipped with Algorithm 7 when $k = 24$. The "INF" means that the adaptive sample size exceeds 10^{10}. In Table VIII, DPColor and DPColorPath are faster than DPTriPath on Stanfordand LiveJournal, and slower on Skitter and Orkut. This is because the clique density differs on these datasets. In Table VIII, ρ_t is much larger than ρ_c and ρ_p on Skitter and Orkut, and they are similar on Stanford and LiveJournal. For example, it has $\rho_c = 0.00002, \rho_p = 0.001, \rho_t = 0.005$ on Skitter and $\rho_c = 0.37, \rho_p = 0.46, \rho_t = 0.88$ on Stanford. These results further confirm the analysis in Section VI-C.

IX. FURTHER RELATED WORK

K-clique and triangle counting: Except the practical algorithms introduced above, there also exist some theoretical studies

TABLE VI
AFFECT OF DIFFERENT THRESHOLD TO SPLIT NETWORK. ($k = 8, t = 5 \times 10^6$)

Datasets	Algs	Time (s): Exact / Sampling / Total			Error (%)	Clique density
		$0.5k$	k	$2k$	$0.5k/k/2k$	$0.5k/k/2k$
Skitter	DPColor	2.7/0.6/3.3	3.6/0.5/4.2	5.3/0.5/5.8	0.47/0.29/0.29	0.05/0.05/0.06
	DPColorPath	2.7/1.0/3.6	3.6/0.7/4.3	5.3/0.4/5.8	0.17/0.13/0.10	0.26/0.26/0.26
	DPTriPath	2.7/3.5/6.2	3.6/3.0/6.6	5.3/2.4/7.7	0.14/0.10/0.04	0.48/0.48/0.48
Google	DPColor	0.3/0.5/0.7	0.5/0.4/0.8	0.5/0.3/0.9	0.14/0.04/0.02	0.90/0.91/0.92
	DPColorPath	0.3/0.4/0.6	0.5/0.2/0.7	0.5/0.2/0.7	0.14/0.04/0.02	0.94/0.94/0.94
	DPTriPath	0.3/0.7/1.0	0.5/0.4/0.9	0.5/0.2/0.8	0.06/0.02/0.02	0.96/0.96/0.95
Orkut	DPColor	108.2/4.1/112.4	138.8/2.3/141.1	224.4/1.2/225.6	97.92/93.95/93.89	0.00/0.00/0.00
	DPColorPath	108.1/50.2/158.3	138.6/37.5/176.1	224.9/22.3/247.1	5.38/5.27/3.67	0.03/0.03/0.03
	DPTriPath	109.0/130.8/239.8	136.6/114.9/251.5	224.6/86.0/310.6	0.25/0.20/0.20	0.23/0.23/0.23

TABLE VII
PERFORMANCE OF THE ADAPTIVE SAMPLING TECHNIQUE IN ALGORITHM 7 FOR VARIOUS DENSITY ($\delta = 0.01$)

Density	Parameters	ϵ	T	Error
0.86	DPColor, $k = 12$, Google	0.01	320358	0.000
		0.05	7254	0.001
		0.1	2563	0.003
0.52	DPColorPath, $k = 15$, Friendster	0.01	273610	0.000
		0.05	20550	0.000
		0.1	10000	0.003
0.16	DPTriPath, $k = 9$, Orkut	0.01	1699744	0.002
		0.05	37527	0.002
		0.1	12945	0.066
0.06	DPTriPath, $k = 15$, Skitter	0.01	2464188	0.001
		0.05	103032	0.002
		0.1	33517	0.003
0.0067	DPColor, $k = 12$, Skitter	0.01	25089143	0.004
		0.05	1672969	0.002
		0.1	314110	0.009
0.0002	DPColorPath, $k = 15$, Orkut	0.01	1120373294	0.002
		0.05	38053125	0.009
		0.1	10222666	0.021

TABLE VIII
COMPARE THE RUNNING TIME OF DPCOLOR, DPCOLORPATH, DPTRIPATHWITH ADAPTIVE SAMPLE SIZE ($\delta = 0.01, k = 24$)

Datasets	ϵ	Time (s)		
		DPColor	DPColorPath	DPTriPath
Stanford	0.01	1.21	0.94	2.84
	0.05	0.44	0.45	1.76
	0.1	0.41	0.41	1.75
Skitter	0.01	INF	100.99	48.43
	0.05	INF	8.95	8.89
	0.1	INF	7.19	6.36
Orkut	0.01	INF	INF	INF
	0.05	INF	INF	822.02
	0.1	INF	INF	308.60
LiveJournal	0.01	26.25	27.52	172.16
	0.05	24.93	25.88	139.91
	0.1	24.90	25.85	138.29

on the k-clique counting problem [37], [38], [39], [40]. Most of these theoretical work focus mainly on devising an algorithm to achieve a better worst-case time complexity. The practical performance of such algorithms is often much worse than the state-of-the-art practical algorithms [16]. Triangle is a specific k-clique for $k = 3$. The problem of counting triangles in a graph has a long history. There are many algorithms in the literature [31], [41], [42], [43], [44]. For example, both [41] and [42] are ordering-based exact triangle counting algorithms. Chu and Cheng [43] developed an I/O-efficient algorithm exact algorithm for triangle listing. Tsourakakis et al. [31] proposed an edge sampling algorithm to approximate the number of triangles in a graph. Becchetti et al. [44] presented an approximate triangle counting algorithm in the semi-streaming model. Tom et al. [45] and Hu et al. [46] developed efficient GPU-parallel algorithms for triangle counting in the shared-memory many-core platforms.

Motif counting: Many exact and sampling-based approximation algorithms have been proposed for motif counting [18], [23], [26], [35], [47], [48]; and some of them can also be used to count k-cliques. Notable example include the color coding based algorithms [23], [26], and edge sampling based algorithms [18]. However, as shown in [20], all these algorithms cannot scale for large graphs and also their practical performance is worse than TuranShadow.

X. CONCLUSION

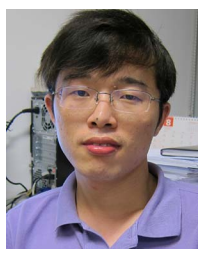
In this paper, we propose a time and space efficient framework for k-clique counting. Our framework first divides the graph into sparse and dense regions based on the average degree. Then, for the sparse regions, we use the state-of-the-art PIVOTER algorithm to compute the exact number of k-cliques. For the dense regions, we develop three novel DP-based k-color set, k-color path, and k-triangle path sampling techniques to estimate the k-clique count, respectively. Extensive experiments on 8 real-life graphs show that our algorithms are very efficient and accurate and also use less space than the state-of-the-art algorithms.

REFERENCES

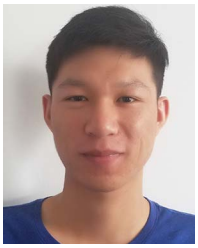
[1] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon, "Network motifs: Simple building blocks of complex networks," *Science*, vol. 298, no. 5594, pp. 763–764, 2010.
[2] S. R. Burt, "Structural holes and good ideas," *Amer. J. Sociol.*, vol. 110, no. 2, pp. 349–399, 2004.
[3] K. Faust, "A puzzle concerning triads in social networks: Graph constraints and the triad census," *Soc. Netw.*, vol. 32, no. 3, pp. 221–233, 2010.
[4] N. Przulj, D. G. Corneil, and I. Jurisica, "Modeling interactome: Scale-free or geometric?," *Bioinformatics*, vol. 20, no. 18, pp. 3508–3515, 2004.
[5] C. Seshadhri and S. Tirthapura, "Scalable subgraph counting: The methods behind the madness," in *Proc. Int. Conf. World Wide Web*, 2019, pp. 1317–1318.

[6] J. W. Berry, B. Hendrickson, R. A. Laviolette, and C. A. Phillips, "Tolerating the community detection resolution limit with edge weighting," *Phys. Rev. E. Stat. Nonlinear Soft Matter Phys.*, vol. 83, no. 5, 2011, Art. no. 056119.
[7] B. Sun, M. Danisch, T. H. Chan, and M. Sozio, "KClist : A simple algorithm for finding k-clique densest subgraphs in large graphs," in *Proc. VLDB Endowment*, vol. 13, no. 10, pp. 1628–1640, 2020.
[8] C. E. Tsourakakis, "The k-clique densest subgraph problem," in *Proc. Int. Conf. World Wide Web*, 2015, pp. 1122–1132.
[9] A. E. Sariyüce, C. Seshadhri, A. Pinar, and Ü. V. Çatalyürek, "Finding the hierarchy of dense subgraphs using nucleus decompositions," in *Proc. Int. Conf. World Wide Web*, 2015, pp. 927–937.
[10] A. R. Benson, D. F. Gleich, and J. Leskovec, "Higher-order organization of complex networks," *Science*, vol. 353, no. 6295, pp. 163–166, 2016.
[11] H. Yin, A. R. Benson, and J. Leskovec, "Higher-order clustering in networks," *Phys. Rev. E*, vol. 97, no. 5, 2017, Art. no. 052306.
[12] N. Chiba and T. Nishizeki, "Arboricity and subgraph listing algorithms," *SIAM J. Comput.*, vol. 14, no. 1, pp. 210–223, 1985.
[13] I. Finocchi, M. Finocchi, and E. G. Fusco, "Clique counting in MapReduce: Algorithms and experiments," *ACM J. Exp. Algorithmics*, vol. 20, pp. 1.7:1–1.7: 20, 2015.
[14] K. Makino and T. Uno, "New algorithms for enumerating all maximal cliques," in *Proc. 9th Scand. Workshop Algorithm Theory*, 2004, pp. 260–272.
[15] M. Danisch, O. Balalau, and M. Sozio, "Listing k-cliques in sparse real-world graphs," in *Proc. Int. Conf. World Wide Web*, 2018, pp. 589–598.
[16] R. Li, S. Gao, L. Qin, G. Wang, W. Yang, and J. X. Yu, "Ordering heuristics for k-clique listing," in *Proc. VLDB Endowment*, vol. 13, no. 11, pp. 2536–2548, 2020.
[17] S. Jain and C. Seshadhri, "The power of pivoting for exact clique counting," in *Proc. ACM Int. Conf. Web Search Data Mining*, 2020, pp. 268–276.
[18] M. Rahman, M. A. Bhuiyan, and M. A. Hasan, "Graft: An efficient graphlet counting method for large graph analysis," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 10, pp. 2466–2478, Oct. 2014.
[19] N. Alon, R. Yuster, and U. Zwick, "Color-coding: A new method for finding simple paths, cycles and other small subgraphs within large graphs," in *Proc. 26th Annu. ACM Symp. Theory Comput.*, 1994, pp. 326–335.
[20] S. Jain and C. Seshadhri, "A fast and provable method for estimating clique counts using Turán's theorem," in *Proc. Int. Conf. World Wide Web*, 2017, pp. 441–449.
[21] D. W. Matula and L. L. Beck, "Smallest-last ordering and clustering and graph coloring algorithms," *J. ACM*, vol. 30, no. 3, pp. 417–427, 1983.
[22] E. Tomita, A. Tanaka, and H. Takahashi, "The worst-case time complexity for generating all maximal cliques and computational experiments," *Theor. Comput. Sci.*, vol. 363, no. 1, pp. 28–42, 2006.
[23] M. Bressan, S. Leucci, and A. Panconesi, "Motivo: Fast motif counting via succinct color coding and adaptive sampling," in *Proc. VLDB Endowment*, vol. 12, no. 11, pp. 1651–1663, 2019.
[24] M. Jha, C. Seshadhri, and A. Pinar, "Path sampling: A fast and provable method for estimating 4-vertex subgraph counts," in *Proc. Int. Conf. World Wide Web*, 2015, pp. 495–505.
[25] P. Wang et al., "MOSS-5: A fast method of approximating counts of 5-node graphlets in large graphs," *IEEE Trans. Knowl. Data Eng.*, vol. 30, no. 1, pp. 73–86, Jan. 2018.
[26] M. Bressan, F. Chierichetti, R. Kumar, S. Leucci, and A. Panconesi, "Motif counting beyond five nodes," *ACM Trans. Knowl. Discov. Data*, vol. 12, no. 4, pp. 48:1–48:25, 2018.
[27] S. Jain and C. Seshadhri, "Provably and efficiently approximating near-cliques using the turán shadow: PEANUTS," in *Proc. Web Conf.*, Taipei, Taiwan, 2020, pp. 1966–1976.
[28] B. Balasundaram and S. Butenko, "Graph domination, coloring and cliques in telecommunications," in *Handbook of Optimization in Telecommunications*. Berlin, Germany: Springer, 2006, pp. 865–890.
[29] L. Chang and L. Qin, "Cohesive subgraph computation over large sparse graphs," in *Proc. IEEE Int. Conf. Data Eng.*, 2019, pp. 2068–2071.
[30] V. Batagelj and M. Zaversnik, "An O(m) algorithm for cores decomposition of networks," 2003, *arXiv:cs/0310049*.
[31] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos, "DOULION: Counting triangles in massive graphs with a coin," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2009, pp. 837–846.
[32] W. Hasenplaugh, T. Kaler, T. B. Schardl, and C. E. Leiserson, "Ordering heuristics for parallel graph coloring," in *Proc. 26th ACM Symp. Parallelism Algorithms Architectures*, 2014, pp. 166–177.
[33] L. Yuan, L. Qin, X. Lin, L. Chang, and W. Zhang, "Effective and efficient dynamic graph coloring," in *Proc. VLDB Endowment*, vol. 11, no. 3, pp. 338–351, 2017.
[34] C.-E. Sarndal, B. Swensson, and J. Wretman, "Model assisted survey sampling," *Springer Sci. Bus. Media*, 2003.
[35] P. Ribeiro, P. Paredes, M. E. P. Silva, D. Aparício, and F. M. A. Silva, "A survey on subgraph counting: Concepts, algorithms, and applications to network motifs and graphlets," *ACM Comput. Surveys*, vol. 54, no. 2, pp. 28:1–28:36, 2022.
[36] M. Almasri, I. E. Hajj, R. Nagi, J. Xiong, and W. Hwu, "Parallel K-clique counting on GPUs," in *Proc. Int. Conf. Supercomputing*, 2022, pp. 21:1–21:14.
[37] T. Eden, D. Ron, and C. Seshadhri, 'On approximating the number of k-cliques in sublinear time," in *Proc. 50th Annu. ACM SIGACT Symp. Theory Comput.*, 2018, pp. 722–734.
[38] T. Eden, D. Ron, and C. Seshadhri, "Faster sublinear approximation of the number of k-cliques in low-arboricity graphs," in *Proc. Symp. Discrete Algorithms*, 2020, pp. 1467–1478.
[39] K. Censor-Hillel, Y. Chang, F. L. Gall, and D. Leitersdorf, "Tight distributed listing of cliques," in *Proc. Symp. Discrete Algorithms*, 2021, pp. 2878–2891.
[40] L. Gianinazzi, M. Besta, Y. Schaffner, and T. Hoefler, "Parallel algorithms for finding large cliques in sparse graphs," in *Proc. 33rd ACM Symp. Parallelism Algorithms Architectures*, 2021, pp. 243–253.
[41] M. Latapy, "Main-memory triangle computations for very large (sparse (power-law)) graphs," *Theor, Comput. Sci.*, vol. 407, no. 1/3, pp. 458–473, 2008.
[42] M. Ortmann and U. Brandes, "Triangle listing algorithms: Back from the diversion," in *Proc. 16th Workshop Algorithm Eng. Experiments*, 2014, pp. 1–8.
[43] S. Chu and J. Cheng, "Triangle listing in massive networks and its applications," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2011, pp. 672–680.
[44] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis, "Efficient semi-streaming algorithms for local triangle counting in massive graphs," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2008, pp. 16–24.
[45] A. S. Tom et al., "Exploring optimizations on shared-memory platforms for parallel triangle counting algorithms," in *Proc. High Perform. Extreme Comput. Conf.*, 2017, pp. 1–7.
[46] L. Hu, L. Zou, and Y. Liu, "Accelerating triangle counting on GPU," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2021, pp. 736–748.
[47] N. Pashanasangi and C. Seshadhri, "Efficiently counting vertex orbits of all 5-vertex subgraphs, by EVOKE," in *Proc. ACM Int. Conf. Web Search Data Mining*, 2020, pp. 447–455.
[48] A. Pinar, C. Seshadhri, and V. Vishal, "ESCAPE: Efficiently counting all 5-vertex subgraphs," in *Proc. Int. Conf. World Wide Web*, 2017, pp. 1431–1440.

Xiaowei Ye received the BE degree from Shandong University, China, in 2021. He is currently working toward the PhD degree with the Beijing Institute of Technology (BIT), Beijing, China. His research interests include subgraph counting, graph data mining and social network analysis.



Rong-Hua Li received the PhD degree from the Chinese University of Hong Kong, in 2013. He is currently a professor with the Beijing Institute of Technology (BIT), Beijing, China. Before joining BIT in 2018, he was an assistant professor with Shenzhen University. His research interests include graph data management and mining, social network analysis, graph computation systems, and graph-based machine learning.



Qiangqiang Dai is currently working toward the PhD degree with the Beijing Institute of Technology (BIT), Beijing, China. His research interests include graph data management and mining, social network analysis, and graph computation systems.

Hongzhi Chen received the PhD degree from the Department of Computer Science and Engineering, Chinese University of Hong Kong, in 2020. He is currently a senior R.D. with ByteDance Infrastructure Team, Beijing, China, working on graph related storage, processing and training systems. His research interests cover the broad area of distributed systems and databases, with special emphasis on graph systems and machine learning/deep learning systems.

Guoren Wang received the BS, MS, and PhD degrees from the Department of Computer Science, Northeastern University, China, in 1988, 1991, and 1996, respectively. Currently, he is a professor with the Beijing Institute of Technology (BIT), Beijing, China. His research interests include graph data management and mining, query processing and optimization, graph computation systems.