Lightning Fast and Space Efficient k-clique Counting

Xiaowei Ye
Beijing Institute of Technology
Beijing, China
yexiaowei@bit.edu.cn

Rong-Hua Li
Beijing Institute of Technology
Beijing, China
lironghuabit@126.com

Qiangqiang Dai
Beijing Institute of Technology
Beijing, China
qiangd66@gmail.com

Hongzhi Chen
ByteDance
Beijing, China
chenhongzhi@bytedance.com

Guoren Wang
Beijing Institute of Technology
Beijing, China
wanggrbit@126.com

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 two 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 sampling-based techniques to estimate the number of k-cliques in the *dense regions* of the graph. Specifically, we develop two 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.

CCS CONCEPTS

• **Networks → Data path algorithms**; • **Theory of computation → Randomness, geometry and discrete structures**.

KEYWORDS

k-clique counting, cohesive subgraphs, graph sampling

ACM Reference Format:
Xiaowei Ye, Rong-Hua Li, Qiangqiang Dai, Hongzhi Chen, and Guoren Wang. 2022. Lightning Fast and Space Efficient k-clique Counting . In *Proceedings of the ACM Web Conference 2022 (WWW '22), April 25–29, 2022, Virtual Event, Lyon, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3485447.3512167

1 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 [31]. Counting the motifs is a fundamental tool in many network analysis applications, including social network analysis, community detection, and bioinformatics [10, 18, 31, 35, 38]. Perhaps the most elementary motif in a graph is the k-clique which has been widely used in a variety of network analysis applications [7, 10, 31, 39, 42].

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. [37] 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 [42] 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 [6, 45].

Motivated by the above applications, many practical k-clique counting algorithms have been proposed [2, 13, 15, 19, 23, 24, 28, 29, 36]. Existing k-clique counting algorithms can be classified into (1) exact k-clique counting methods, and (2) sampling-based approximation solutions. Chiba and Nishizeki [13] 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. [19] based on a degree ordering technique. Subsequently, Danisch et al. [15] further improved this algorithm by using a degeneracy ordering technique [30]. More recently, Li et al. [28] 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

WWW '22, April 25–29, 2022, Virtual Event, Lyon, France
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9096-5/22/04…$15.00
https://doi.org/10.1145/3485447.3512167

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 [41]. 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. PIVOTER was shown to be much faster than previous k-clique enumeration based algorithms [13, 15, 19, 28, 29].

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 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 maximal cliques (with complicated overlap relationships), resulting in a large search tree of Pivoter (e.g., see the results on the LiveJournal dataset in [24]).

Approximation solutions based on sampling are typically able to handle large dense graphs when k is not very large [9, 23, 36]. However, to achieve a desired accuracy, previous sampling-based solutions either require a huge number of samples [26, 36, 44] or consume very high storage space [2, 8, 9, 23] for a relatively large k (e.g., $k \geq 8$). The state-of-the-art sampling-based approximation algorithm is the TuranShadow algorithm which was proposed by Jain and Seshadhri [23]. As shown in [23], TuranShadow is much faster and more accurate than the other 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 [13]. Therefore, on large graphs, TuranShadow is very costly for a large k.

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 two novel sampling technique based on a concept of graph coloring [3] 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 DP-based k-color path sampling technique to further improve the efficiency and accuracy. Here a k-color path is a connected k-color set which is more effective to approximate a k-clique. Moreover, unlike TuranShadow, both of our sampling-based solutions take near-linear space w.r.t. the graph size. In summary, we make the following contributions.

New algorithmic framework. 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.

Two novel sampling algorithms. We develop two 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.

Extensive experiments. 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 (TuranShadow) 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× ∼ 14× speedup ratios when using 16 threads in our experiments. For reproducibility purpose, the source code of this paper is released at https://github.com/LightWant/dpcolor.

2 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, 23, 24, 28].

Let $\pi : V \longrightarrow \{v_1, ..., v_n\}$ 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.

Algorithm 1: 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 $ans \leftarrow ans$ + PIVOTER($N_v(\vec{G})$, $k - 1$);
5      else $S \leftarrow S \cup \{v\}$;
6  return $ans$ + Sampling($\vec{G}$, $S$, $k$, $t$);
```

Note that many different ordering heuristics for k-clique counting have been developed in the literature [28]. Among them, a widely-used ordering heuristics is the degeneracy ordering [30], where the degeneracy is a metric to measure the sparsity of a graph [12]. Specifically, the degeneracy ordering of nodes in G is defined as an ordering $\{v_1, ..., v_n\}$ such that the degree of v_i is minimum in the subgraph of G induced by $\{v_i, ..., v_n\}$ 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 [4]. 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 [12, 30], the degeneracy ordering based k-clique counting algorithms are often very efficient in practice [28]. In this work, we will also use the degeneracy ordering to design our algorithms.

3 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 1.

Note that in Algorithm 1, 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 1, 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).

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 [36, 43], are often inefficient, because those algorithms require a considerable number of samples to achieve a desired accuracy [23]. The color-coding based techniques often consume a significant number of space [2, 8, 9] and also they are less efficient than the TuranShadow algorithm [23]. The TuranShadow algorithm [23], which is the state-of-the-art sampling-based technique, also needs much space to store the Tur̗an Shadow. Moreover, the construction time of the Tur̗an Shadow is often very long for large graphs, because the worst-case time complexity of TuranShadow is exponential. In Sections 4 and 5, we will propose two novel sampling algorithms to tackle this problem.

Parallel implementation. Note that the proposed framework (Algorithm 1) 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 1, 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.

4 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 [3, 21, 46]. 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 $[1, \cdots, \chi]$ 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 [3], we use a linear-time greedy coloring algorithm [21, 46] to obtain a feasible coloring solution. Based on a feasible coloring solution, we define a concept called k-color set as follows.

Definition 4.1. 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 4.1, the nodes of any k-clique must form a k-color set. In particular, we have the following lemma.

Lemma 4.2. *Given a graph G, all k-cliques must be contained in the set of all k-color sets.*

Let $\mathrm{cnt}_k(G, clique)$ and $\mathrm{cnt}_k(G, color)$ be the number of k-cliques and k-color sets of G respectively. Denoted by ρ_k 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., $\rho_k = \frac{\mathrm{cnt}_k(G, clique)}{\mathrm{cnt}_k(G, color)}$. 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_k

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_k.

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_k, 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.

4.1 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 [21, 46]. 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 Eq. (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 Eqs. (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 2 (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.

Algorithm 2: DPSampler (G, χ, k)

```
    Input: A colored graph $G = (V, E)$, an integer $k$, and the maximum color number $\chi$
    Output: A uniformly sampled $k$-color set
1   $F \leftarrow$ DPCount($G$, $\chi$, $k$);
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($G$, $\chi$, $k$)
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 $\chi$ do $F(i, 0) = 1$;
9       foreach $i = 1$ to $\chi$ 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 ($G$, $P$, $R$, $i-1$, $j$) ;
```

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 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 Eq. (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 Eq. (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 2.

Algorithm 2 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 Eq. (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 2.

Lemma 4.3. *The* DPSampling *procedure in Algorithm 2 outputs a k-color set of G if $\chi \geq k$.*

Theorem 4.4. *Algorithm 2 outputs a uniform k-color set.*

The following theorem shows the complexity of Algorithm 2.

Theorem 4.5. *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 2 are $O(\chi k)$.*

Algorithm 3: 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 [21, 46];
2  Let $\chi$ be the number of colors obtained;
3  foreach $v \in S$ do
4      $F_v \leftarrow$ 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      Independently sample a node $v$ from $D$;
10     $R \leftarrow \{v\} \cup$ DPSampler($G(N_v(\vec{G}))$, $\chi$, $k-1$);
11     if $R$ is a $k$-clique then
12         $successTimes \leftarrow successTimes + 1$;
13 $\rho_k \leftarrow successTimes/t$;
14 return $\rho_k \times cntKCol$;
```

4.2 Estimating the number of k-cliques

By Theorem 4.4, we can first make use of Algorithm 2 to uniformly sample k-color sets from G, and then estimate the clique density ρ_k in the k-color sets of G. After that, the number of k-cliques in G can be estimated by $\rho_k \times F(\chi, k)$. 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 3.

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 3 first colors the graph using a linear-time greedy algorithm [21, 46] (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 3 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_k in the sampled k-color sets (lines 11-13), and then estimates the k-clique count as $\rho_k \times cntKCol$ (line 14). The following theorem shows that Algorithm 3 can obtain an unbiased estimator.

Theorem 4.6. *Algorithm 3 outputs an unbiased estimator for the number of k-cliques in the dense regions of G.*

By applying the classic Chernoff bound, we can easily derive that Algorithm 3 is able to produce a $1 - \epsilon$ approximation of the k-clique count in the dense regions of the graph.

Theorem 4.7. *Algorithm 3 returns a $1 - \epsilon$ approximation of the number of k-cliques in the dense regions of G with probability $1 - 2\sigma$ if $t \geq \frac{3}{\rho_k \epsilon^2} ln \frac{1}{\sigma}$, where ϵ and σ are small positive values and t is the sample size.*

Note that by Theorem 4.7, the sample size of our algorithm relies on the k-clique density ρ_k. Since ρ_k is often not very small in the dense regions of a graph, Algorithm 3 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 3.

Theorem 4.8. *Algorithm 3 consumes $O((|S| + t)\chi k + k^2 t + m + n)$ time and $O(m + n + \chi k)$ space.*

Remark. The proposed k-color set sampling algorithm is completely different from the traditional color coding technique [2, 8, 9] 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 $\frac{k!}{k^k}$ [2]. 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.

5 CONNECTED k-COLOR SET SAMPLING

Recall that to achieve a $1 - \epsilon$ approximation, the sample size of Algorithm 3 heavily relies on the k-clique density over the k-color sets, i.e., ρ_k (see Theorem 4.7). Although the dense regions of a graph G often have a relatively high ρ_k, 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.

For any k-color set in G, we can observe that it 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 4 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.

5.1 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 [28]. 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 5.1. *Let $\vec{G}$ be the DAG generated by the color ordering. Then, any $(k-1)$-path in $\vec{G}$ forms a k-color path.*

Theorem 5.2. *Let $\vec{G}$ be the DAG generated by the color ordering. Then, any k-clique $C = \{v_1, v_2, \cdots, v_k\}$ in G is a k-color path in $\vec{G}$.*

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. To estimate the number of k-cliques in G, we need to compute ρ_p and the number of k-color paths as well. Below, we propose a DP algorithm to achieve this goal.

Let $\vec{G}_{v_i}$ be a subgraph of $\vec{G}$ induced by $\{v_i, ..., v_n\}$. 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:

$$\mathsf{cnt}_{k-1}(\vec{G}, path) = \sum_{v_i \in \vec{G}} H(v_i, k-1). \tag{4}$$

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:

$$H(v_i, j) = \sum_{v_x \in N_{v_i}(\vec{G}_{v_i})} H(v_x, j-1). \tag{5}$$

Initially, we have

$$\begin{cases} H(v_i, 0) = 1, & \text{for all } i \in [1, n], \\ H(v_i, j) = 0, & \text{for all } i \in [1, n], j \in [1, k-1]. \end{cases} \tag{6}$$

Based on Eqs. (4), (5) and (6), 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 4 (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 4.1, 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 (Eq. (5)), 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 Eq. (5). We can recursively perform this sampling procedure to obtain a k-color path. The detailed implementation of this sampling technique is shown in Algorithm 4.

Algorithm 4: 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         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 $R$;
```

Algorithm 4 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 4 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 Eq. (5) (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 4 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 4 will be executed k times which results in a k-color path. The following theorem shows that Algorithm 4 can obtain a uniform k-color path.

Theorem 5.3. *Algorithm 4 outputs a uniform k-color path.*

We analyze the time and space complexity of Algorithm 4 in the following theorem.

Theorem 5.4. *Given an input graph G with n nodes and m edges, Algorithm 4 takes $O(\chi nk + m)$ time and uses $O(kn + m)$ space, where χ is the maximum color value.*

5.2 Estimating the k-clique counts

Based on Algorithm 4, 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 3 by (1) replacing the DPCount procedure in line 4 of Algorithm 3 with the DPPathCount procedure, and (2) replacing DPSampling in line 10 of Algorithm 3 with DPPathSampling. Due to the space limit, we omit the details of this modified algorithm. Similar to Theorems 4.6 and 4.7, the estimator based on the k-color path sampling is also

Table 1: Datasets

Networks	n	m	δ
webStanford	281,903	1,992,636	71
DBLP	425,957	1,049,866	113
webBerkStan	685,230	6,649,470	201
webGoogle	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

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 3, because $\rho_p \geq \rho_k$.

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.

6 EXPERIMENTS

6.1 Experimental setup

We compare the proposed algorithms with three state-of-the-art k-clique counting algorithms which are kClist [15, 28], PIVOTER [24], TuranShadow [23]. 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. [28] 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 Seshadhri [23, 24]. PEANUTS [25] is an improved version of TuranShadow by the technique of Prefixed-Shadow, thus we use PEANUTS as the baseline instead of TuranShadow. The C++ codes of all these three algorithms are publicly available, thus we use their implementations in our experiments. For our algorithms, we implement DPColor and DPColorPath. DPColor denotes Algorithm 1 integrated with the k-color set sampling algorithm (Algorithm 2 and Algorithm 3), while DPColorPath is Algorithm 1 equipped with the k-color path sampling technique (Algorithm 4). Both DPColor and DPColorPath are implemented in C++. All algorithms are evaluated on a PC with two 2.1 GHz Xeon CPUs (16 cores in total) and 128GB memory running CentOS 7.6.

Datasets. We use 8 large real-life datasets in our experiments. Table 1 summarizes the detailed statistic information of all datasets. The last column of Table 1 denotes the degeneracy of the graph. The webStanford, webBerkStan, and webGoogle datasets are web graphs. DBLP is a co-authorship network, and Skitter is an internet graph. Orkut, LiveJournal, and Friendster are social networks. All datasets are downloaded from (snap.stanford.edu).

6.2 Experimental results

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, and DPColorPath), 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. 1 shows the running time of different algorithms with varying k.

We first compare our algorithms with kClist and PIVOTER. As can be seen, both DPColor and DPColorPath 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 [1], where $k \leq 8$, to compute the relative errors for the approximation algorithms. Moreover, as reported in [1], 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. These results indicate that our algorithms are extremely efficient for k-clique counting.

By comparing our algorithms with PEANUTS, we can see that both DPColor and DPColorPath are 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, both DPColor and DPColorPath take around 0.1 second, 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 is still very efficient on these two datasets. For our algorithms, DPColorPath is generally faster than DPColor. Moreover, the performance of DPColorPath is much more stable than DPColor on all datasets. These results confirm our theoretic analysis in Sections 4 and 5.

Relative errors with varying sample size. Fig. 2 shows the relative errors of three algorithms with varying k on webStanford and LiveJournal. Similar results can also be observed on the other datasets. As shown in Fig. 2, the relative error of DPColorPath is consistently lower than those of DPColor and PEANUTS with the same sample size. In general, the relative errors of all algorithms decrease with the sample size increases. Moreover, we can see that DPColorPath obtain a 10^{-5} relative error on all datasets when the sample size is 10^8, indicating that DPColorPath can achieve very high accuracy using a reasonable number of samples. These results further confirm the efficiency and effectiveness of our techniques.

K-clique density. In this experiment, we evaluate the k-clique densities over the k-color sets (ρ_k) and the k-color paths (ρ_p) in the dense regions of the graph, respectively. The results on all datasets are reported in Table 2. As expected, ρ_p is no larger than ρ_k on all datasets. Moreover, both ρ_k and ρ_p can achieve a very high value on most datasets. For example, on DBLP and webBerkStan, both

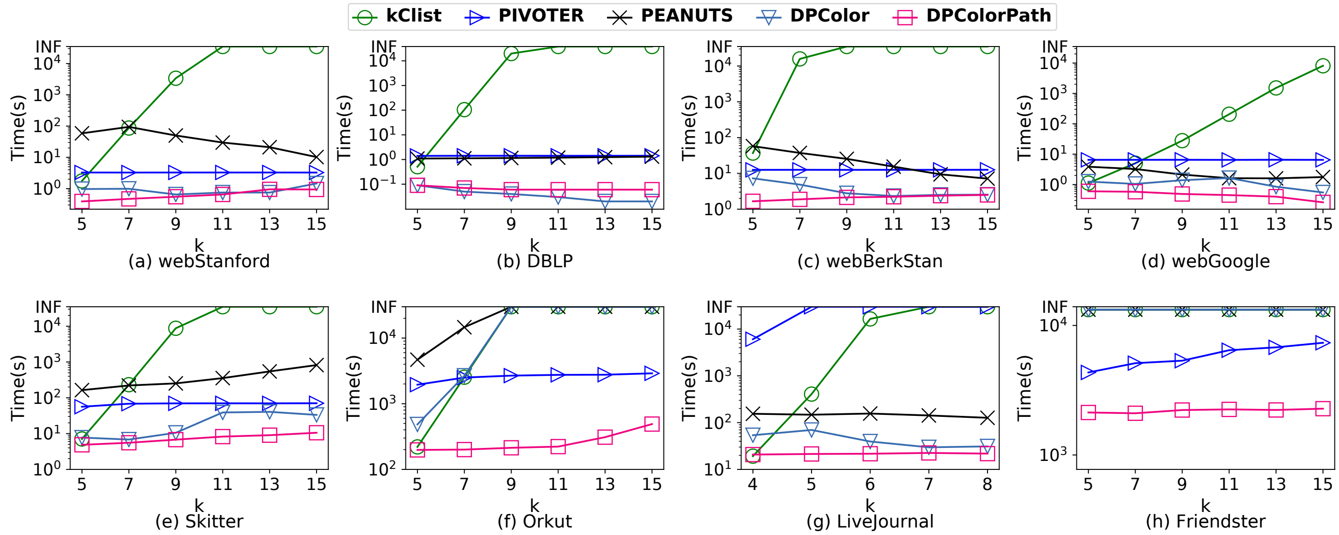
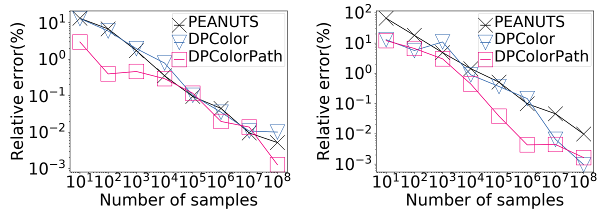


Figure 1: Running time of different algorithms (the relative errors for PEANUTS, DPColor, DPColorPath are set to 0.1%)



(a) webStanford (b) LiveJournal

Figure 2: Relative errors with varying sample size ($k = 8$)

Table 2: The *k-clique* densities (ρ_k/ρ_p) in the dense regions (%)

Networks	$k = 3$	$k = 8$	$k = 12$	$k = 15$
webStanford	83.0/100.0	70.8/77.7	54.3/61.2	45.8/53.1
DBLP	97.2/100.0	100.0/100.0	100.0/100.0	100.0/100.0
webBerkStan	94.7/100.0	99.9/100.0	100.0/100.0	100.0/100.0
webGoogle	94.6/100.0	91.2/94.4	86.1/87.8	84.7/85.7
Skitter	42.3/100.0	5.3/26.4	0.7/6.4	0.1/2.3
Orkut	20.5/100.0	0.0/2.6	0.0/0.2	0.0/0.0002
LiveJournal	54.3/100.0	80.4/91.0	-/-	-/-
Friendster	10.1/100.0	0.0/18.1	0.0/55.6	0.0/52.2

ρ_k and ρ_p are near to 100%. In general, both ρ_k and ρ_p decreases with k increases. Nevertheless, on most datasets, ρ_p is always very large even when $k = 15$. These results further confirm that the proposed techniques can achieve high accuracy on real-life graphs.

Memory overheads. Fig. 3 shows the memory usages of various algorithms on webBerkStan 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 higher than the other algorithms, as it needs to store the 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.

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 both DPColor and DPColorPath using OpenMP. We fix the sample size as 5×10^6 to evaluate the runtime of DPColor and DPColorPath on the two largest datasets. The results are shown in Table 3. As can be seen, both DPColor and DPColorPath perform very well. Both DPColor and DPColorPath can achieve 12× ∼ 14× speedups when using 16 threads. This result indicates a high degree of parallelism of our algorithms.

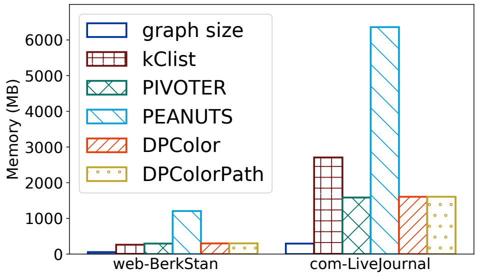


Figure 3: Memory usage of various algorithms ($k = 8$)

Table 3: Runtime of our parallel algorithms ($k = 8, t = 5{\times}10^6$)

Threads	DPColor (sec.)		DPColorPath (sec.)	
	LiveJournal	Friendster	LiveJournal	Friendster
1	24.8	2481.5	28.4	2132.2
4	7.1	650.3	7.5	559.6
8	4.3	341.6	3.9	293.3
12	2.7	244.4	2.7	210.3
16	2.1	196.5	2.1	171.9

7 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 two novel DP-based k-color set and k-color 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.

ACKNOWLEDGMENTS

This work was partially supported by (i) National Key Research and Development Program of China 2020AAA0108503, (ii) NSFC Grants 62072034, U1809206, and 61772346. Rong-Hua Li is the corresponding author of this paper.

REFERENCES

[1] Mohammad Almasri, Izzat El Hajj, Rakesh Nagi, Jinjun Xiong, and Wen-Mei W. Hwu. 2021. K-Clique Counting on GPUs. *CoRR* abs/2104.13209 (2021).

[2] Noga Alon, Raphael Yuster, and Uri Zwick. 1994. Color-coding: a new method for finding simple paths, cycles and other small subgraphs within large graphs. In *STOC*.

[3] Balabhaskar Balasundaram and Sergiy Butenko. 2006. Graph Domination, Coloring and Cliques in Telecommunications. In *Handbook of Optimization in Telecommunications*. Springer, 865–890.

[4] Vladimir Batagelj and Matjaz Zaversnik. 2003. An O(m) Algorithm for Cores Decomposition of Networks. *CoRR* cs.DS/0310049 (2003).

[5] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis. 2008. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *KDD*.

[6] Austin R. Benson, David F. Gleich, and Jure Leskovec. 2016. Higher-order organization of complex networks. *Science* 353, 6295 (2016).

[7] J. W. Berry, B. Hendrickson, R. A. Laviolette, and C. A. Phillips. 2011. Tolerating the Community Detection Resolution Limit with Edge Weighting. *Physical Review E Statistical Nonlinear & Soft Matter Physics* 83, 5 (2011), 056119.

[8] Marco Bressan, Flavio Chierichetti, Ravi Kumar, Stefano Leucci, and Alessandro Panconesi. 2018. Motif Counting Beyond Five Nodes. *ACM Trans. Knowl. Discov. Data* 12, 4 (2018), 48:1–48:25.

[9] Marco Bressan, Stefano Leucci, and Alessandro Panconesi. 2019. Motivo: Fast Motif Counting via Succinct Color Coding and Adaptive Sampling. *Proc. VLDB Endow.* 12, 11 (2019), 1651–1663.

[10] S Ronald Burt. 2004. Structural holes and good ideas. *Amer. J. Sociology* 110, 2 (2004), 349–399.

[11] Keren Censor-Hillel, Yi-Jun Chang, François Le Gall, and Dean Leitersdorf. 2021. Tight Distributed Listing of Cliques. In *SODA*.

[12] Lijun Chang and Lu Qin. 2019. Cohesive Subgraph Computation Over Large Sparse Graphs. In *ICDE*.

[13] Norishige Chiba and Takao Nishizeki. 1985. Arboricity and Subgraph Listing Algorithms. *SIAM J. Comput.* 14, 1 (1985), 210–223.

[14] Shumo Chu and James Cheng. 2011. Triangle listing in massive networks and its applications. In *KDD*.

[15] Maximilien Danisch, Oana Balalau, and Mauro Sozio. 2018. Listing k-cliques in Sparse Real-World Graphs. In *WWW*.

[16] Talya Eden, Dana Ron, and C. Seshadhri. 2018. On approximating the number of k-cliques in sublinear time. In *STOC*.

[17] Talya Eden, Dana Ron, and C. Seshadhri. 2020. Faster sublinear approximation of the number of k-cliques in low-arboricity graphs. In *SODA*.

[18] Katherine Faust. 2010. A puzzle concerning triads in social networks: Graph constraints and the triad census. *Soc. Networks* 32, 3 (2010), 221–233.

[19] Irene Finocchi, Marco Finocchi, and Emanuele G. Fusco. 2015. Clique Counting in MapReduce: Algorithms and Experiments. *ACM J. Exp. Algorithmics* 20 (2015), 1.7:1–1.7:20.

[20] Lukas Gianinazzi, Maciej Besta, Yannick Schaffner, and Torsten Hoefler. 2021. Parallel Algorithms for Finding Large Cliques in Sparse Graphs. In *SPAA*.

[21] William Hasenplaugh, Tim Kaler, Tao B. Schardl, and Charles E. Leiserson. 2014. Ordering heuristics for parallel graph coloring. In *SPAA*.

[22] Lin Hu, Lei Zou, and Yu Liu. 2021. Accelerating Triangle Counting on GPU. In *SIGMOD*.

[23] Shweta Jain and C. Seshadhri. 2017. A Fast and Provable Method for Estimating Clique Counts Using Turán's Theorem. In *WWW*.

[24] Shweta Jain and C. Seshadhri. 2020. The Power of Pivoting for Exact Clique Counting. In *WSDM*.

[25] Shweta Jain and C. Seshadhri. 2020. Provably and Efficiently Approximating Near-cliques using the Turán Shadow: PEANUTS. In *WWW '20: The Web Conference 2020, Taipei, Taiwan, April 20-24, 2020*. 1966–1976.

[26] Madhav Jha, C. Seshadhri, and Ali Pinar. 2015. Path Sampling: A Fast and Provable Method for Estimating 4-Vertex Subgraph Counts. In *WWW*.

[27] Matthieu Latapy. 2008. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theor. Comput. Sci.* 407, 1-3 (2008), 458–473.

[28] Ronghua Li, Sen Gao, Lu Qin, Guoren Wang, Weihua Yang, and Jeffrey Xu Yu. 2020. Ordering Heuristics for k-clique Listing. *Proc. VLDB Endow.* 13, 11 (2020), 2536–2548.

[29] Kazuhisa Makino and Takeaki Uno. 2004. New Algorithms for Enumerating All Maximal Cliques. In *9th Scandinavian Workshop on Algorithm Theory*.

[30] David W. Matula and Leland L. Beck. 1983. Smallest-Last Ordering and clustering and Graph Coloring Algorithms. *J. ACM* 30, 3 (1983), 417–427.

[31] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D Chklovskii, and U. Alon. 2010. Network Motifs: Simple Building Blocks of Complex Networks. *Science* 298, 5594 (2010), 763–764.

[32] Mark Ortmann and Ulrik Brandes. 2014. Triangle Listing Algorithms: Back from the Diversion. In *ALENEX*.

[33] Noujan Pashanasangi and C. Seshadhri. 2020. Efficiently Counting Vertex Orbits of All 5-vertex Subgraphs, by EVOKE. In *WSDM*.

[34] Ali Pinar, C. Seshadhri, and Vaidyanathan Vishal. 2017. ESCAPE: Efficiently Counting All 5-Vertex Subgraphs. In *WWW*.

[35] Natasa Przulj, Derek G. Corneil, and Igor Jurisica. 2004. Modeling interactome: scale-free or geometric? *Bioinform.* 20, 18 (2004), 3508–3515.

[36] Mahmudur Rahman, Mansurul Alam Bhuiyan, and Mohammad Al Hasan. 2014. Graft: An Efficient Graphlet Counting Method for Large Graph Analysis. *IEEE Trans. Knowl. Data Eng.* 26, 10 (2014), 2466–2478.

[37] Ahmet Erdem Sariyüce, C. Seshadhri, Ali Pinar, and Ümit V. Çatalyürek. 2015. Finding the Hierarchy of Dense Subgraphs using Nucleus Decompositions. In *WWW*.

[38] Comandur Seshadhri and Srikanta Tirthapura. 2019. Scalable Subgraph Counting: The Methods Behind The Madness. In *WWW*.

[39] Bintao Sun, Maximilien Danisch, T.-H. Hubert Chan, and Mauro Sozio. 2020. KClist++: A Simple Algorithm for Finding k-Clique Densest Subgraphs in Large Graphs. *Proc. VLDB Endow.* 13, 10 (2020), 1628–1640.

[40] Ancy Sarah Tom, Narayanan Sundaram, Nesreen K. Ahmed, Shaden Smith, Stijn Eyerman, Midhunchandra Kodiyath, Ibrahim Hur, Fabrizio Petrini, and George Karypis. 2017. Exploring optimizations on shared-memory platforms for parallel triangle counting algorithms. In *HPEC*.

[41] Etsuji Tomita, Akira Tanaka, and Haruhisa Takahashi. 2006. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theor. Comput. Sci.* 363, 1 (2006), 28–42.

[42] Charalampos E. Tsourakakis. 2015. The K-clique Densest Subgraph Problem. In *WWW*.

[43] Charalampos E. Tsourakakis, U Kang, Gary L. Miller, and Christos Faloutsos. 2009. DOULION: counting triangles in massive graphs with a coin. In *KDD*.

[44] Pinghui Wang, Junzhou Zhao, Xiangliang Zhang, Zhenguo Li, Jiefeng Cheng, John C. S. Lui, Don Towsley, Jing Tao, and Xiaohong Guan. 2018. MOSS-5: A Fast Method of Approximating Counts of 5-Node Graphlets in Large Graphs. *IEEE Trans. Knowl. Data Eng.* 30, 1 (2018), 73–86.

[45] Hao Yin, Austin R. Benson, and Jure Leskovec. 2017. Higher-order clustering in networks. *Physical Review E* 97, 5 (2017), 052306.

[46] Long Yuan, Lu Qin, Xuemin Lin, Lijun Chang, and Wenjie Zhang. 2017. Effective and Efficient Dynamic Graph Coloring. *Proc. VLDB Endow.* 11, 3 (2017), 338–351.

Table 4: The ratio of the k-cliques in the sparse regions.

Networks	$k = 3$	$k = 8$	$k = 12$	$k = 15$
webStanford	6.15%	0.01%	0.00%	0.00%
DBLP	33.11%	0.00%	0.00%	0.00%
webBerkStan	3.80%	0.00%	0.00%	0.00%
webGoogle	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%

8 SUPPLEMENTARY MATERIALS

8.1 Additional experiments

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, then the PIVOTER algorithm should be more efficient. Table 4 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.

Trade-off between accuracy and sample size. Table 5 shows the sample sizes needed by TuranShadow, DPColor and DPColorPath to meet a desired relative error given that $k = 8$. The results are consistent when setting k to other values. As expected, the sample size required by different algorithms increases with the relative error decreases. Both DPColor and DPColorPath requires less samples than TuranShadow to achieve a desired relative errors on most datasets. Moreover, we can see that in most settings, the sample size of DPColorPath is one order of magnitude less than those of TuranShadow and DPColor. For example, to achieve a 0.01% relative error on LiveJournal, DPColorPath only needs 2×10^5 samples, whereas TuranShadow and DPColor require 3×10^7 and 7×10^6 samples respectively. These results further confirm the superiority of the proposed k-color path sampling technique.

8.2 Missing proofs

The proof of lemma 4.3. 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 Eq. (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. □

The proof of Theorem 4.4. 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:

$$\Pr(X) = \Pr((\sum_{i=1}^{\chi} Y_i) = k). \tag{7}$$

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 $C = \{c_1, ..., c_k\}$, 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 $\{[c_1+1, \chi], [c_1, c_1], [c_2+1, c_1-1], \cdots, [c_k+1, c_{k-1}-1], [c_k, c_k], [1, c_k-1]\}$. Note that DPSampling only selects a color in the sub-intervals $[c_i, c_i]$ for every $i = 1, \cdots, k$, 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

$$\begin{aligned}
&\frac{F(\chi-1,k)}{F(\chi,k)} \times \frac{F(\chi-2,k)}{F(\chi-1,k)} \times \cdots \times \frac{F(c_1,k)}{F(c_1+1,k)} \times \frac{a_{c_1} \times F(c_1-1,k-1)}{F(c_1,k)} \\
&\times \frac{F(c_1-2,k-1)}{F(c_1-1,k-1)} \times \cdots \times \frac{F(c_2,k-1)}{F(c_2+1,k-1)} \times \frac{a_{c_2} \times F(c_2-1,k-2)}{F(c_2,k-1)} \\
&\times \cdots \times \frac{F(c_k,1)}{F(c_k+1,1)} \times \frac{a_{c_k} \times F(c_k-1,0)}{F(c_k,1)} \\
&= \frac{a_{c_1} \times a_{c_2} \times \cdots \times a_{c_k}}{F(\chi,k)}.
\end{aligned} \tag{8}$$

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

$$\begin{aligned}
\Pr(k\text{-color set}) &= \Pr(k \text{ nodes with different colors}|X) \times \Pr(X) \\
&= \frac{1}{a_{c_1} \times a_{c_2} \times \cdots \times a_{c_k}} \times \frac{a_{c_1} \times a_{c_2} \times \cdots \times a_{c_k}}{F(\chi,k)} \\
&= \frac{1}{F(\chi,k)} = \frac{1}{\mathrm{cnt}_k(G, color)}
\end{aligned} \tag{9}$$

By Eq. (9), each k-color set is uniformly sampled, thus the theorem is established. □

The proof of Theorem 4.5. 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 2 is $O(\chi k)$. For the space complexity, Algorithm 2 only requires $O(\chi k)$ additional space to store the DP table F and the probabilities p. □

The proof of Theorem 4.6. Let $X_i = 1$ if the i_{th} sampled k-color set is a k-clique, otherwise $X_i = 0$. Observe that

$$\begin{aligned}
\Pr(X_i = 1) = \sum_{v \in S} [&\Pr(choose\ v\ from\ D) \\
&\times \Pr(choose\ a\ clique\ from\ G(N_v(\vec{G})))].
\end{aligned} \tag{10}$$

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 $\frac{cnt_k(G(N_v(\vec{G})), clique)}{F_v(\chi,k-1)}$. Thus, $Pr(X_i = 1) =$

Table 5: Sample sizes vs. relative errors of TuranShadow, DPColor **and** DPColorPath **(k = 8)**

Networks	**$\epsilon = 1\%$**			**$\epsilon = 0.1\%$**			**$\epsilon = 0.01\%$**		
	TuranShadow	DPColor	DPColorPath	TuranShadow	DPColor	DPColorPath	TuranShadow	DPColor	DPColorPath
webStanford	$5e10^3$	$1e10^4$	$2e10^3$	$1e10^5$	$1e10^5$	$7e10^4$	$1e10^8$	$5e10^6$	$1e10^6$
DBLP	$1e10^1$	$1e10^1$	$1e10^1$	$1e10^1$	$1e10^1$	$1e10^1$	$5e10^3$	$1e10^1$	$1e10^1$
webBerkStan	$1e10^3$	$1e10^1$	$1e10^1$	$1e10^3$	$1e10^1$	$1e10^1$	$5e10^5$	$5e10^5$	$2e10^5$
webGoogle	$1e10^4$	$1e10^3$	$1e10^2$	$1e10^5$	$3e10^4$	$2e10^4$	$4e10^7$	$4e10^5$	$8e10^5$
Skitter	$1e10^3$	$3e10^5$	$1e10^4$	$1e10^6$	$1e10^7$	$5e10^5$	$2e10^7$	$5e10^7$	$4e10^6$
Orkut	$1e10^6$	$>1e10^8$	$3e10^6$	$5e10^6$	$>1e10^8$	$3e10^7$	$1e10^7$	$>1e10^8$	$8e10^7$
LiveJournal	$1e10^4$	$1e10^4$	$3e10^3$	$1e10^5$	$5e10^6$	$1e10^4$	$3e10^7$	$7e10^6$	$2e10^5$
Friendster	–	$>1e10^8$	$5e10^6$	–	$>1e10^8$	$1e10^7$	–	$>1e10^8$	$1e10^7$

$\frac{\sum_{v\in S} cnt_k(G(N_v(\vec{G})), clique)}{\sum_{v\in S} cnt_k(G(N_v(\vec{G})), color)}$. 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

$$
\begin{aligned}
E[cntKCol \times \frac{\sum_{i\le t} X_i}{t}] &= \sum_{v\in S} cnt_k(G(N_v(\vec{G})), color) \times \frac{\sum_{i\le t} E[X_i]}{t} \\
&= \sum_{v\in S} cnt_k(G(N_v(\vec{G})), clique).
\end{aligned} \tag{11}
$$

Therefore, Algorithm 3 returns an unbiased estimator of the k-clique count in the dense regions of G. □

The proof of Theorem 4.7. Denote by $\hat{\rho_k}$ the estimator of the k-clique density (line 13 of Algorithm 3). Since our estimator is unbiased, we have $E[\hat{\rho_k}] = \rho_k$. Then, the expected number of k-cliques in the t samples is $E[\hat{\rho_k} t] = \rho_k t$. Based on the Chernoff bound, we easily obtain the following results:

$$\Pr(\hat{\rho_k} t \le (1-\epsilon)\rho_k t) \le \exp(-\frac{\epsilon^2 \rho_k t}{2}) \le \exp(-\frac{\epsilon^2 \rho_k t}{3}), \tag{12}$$

$$\Pr(\hat{\rho_k} t \ge (1+\epsilon)\rho_k t) \le \exp(-\frac{\epsilon^2 \rho_k t}{3}). \tag{13}$$

Further, we have:

$$\Pr(\frac{|\hat{\rho_k} - \rho_k|}{\rho_k} \ge \epsilon) \le 2\exp(-\frac{\epsilon^2 \rho_k t}{3}). \tag{14}$$

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

The proof of Theorem 4.8. For the time complexity, Algorithm 3 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 3 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 3 is $O(m + n + \chi k)$. □

The proof of Theorem 5.1. Let $P = \{(v_1, v_2), (v_2, v_3), \cdots, (v_{k-1}, v_k)\}$ be a $(k-1)$-path in $\vec{G}$. By the color ordering, we have $c(v_i) \le c(v_{i+1})$ 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 $c(v_i) \ne c(v_{i+1})$ for each $i \in [1, k-1]$. As a result, the path S is a k-color path. □

The proof of Theorem 5.2. Let $C = \{v_1, v_2, \cdots, v_k\}$ be a k-clique in G. Clearly, the nodes in C have different colors. Suppose without loss of generality that $c(v_1) < c(v_2), \cdots, c(v_k)$. Since $\vec{G}$ is generated by the color ordering, there must exist a path $\{(v_1, v_2), \cdots, (v_{k-1}, v_k)\}$ in $\vec{G}$ which also forms a valid k-color path. □

The proof of Theorem 5.3. Consider a path $\{v_1, v_2, \cdots, v_k\}$. Let X be the event of this path being sampled by Algorithm 4. 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

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

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. □

The proof of Theorem 5.4. 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 DPPathSampling procedure 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 4 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. □

8.3 Further related work

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

k-clique counting problem [11, 16, 17, 20]. 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 [28]. 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 [5, 14, 27, 32, 43]. For example, both [27] and [32] are ordering-based exact triangle counting algorithms. Chu and Cheng [14] developed an I/O-efficient algorithm exact algorithm for triangle listing. Tsourakakis et al. [43] proposed an edge sampling algorithm to approximate the number of triangles in a graph. Becchetti et al. [5] presented an approximate triangle counting algorithm in the semi-streaming model. Tom et al. [40] and Hu et al. [22] 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 [8, 9, 33, 34, 36]; and some of them can also be used to count k-cliques. Notable example include the color coding based algorithms [8, 9], and edge sampling based algorithms [36]. However, as shown in [23], all these algorithms cannot scale for large graphs and also their practical performance is worse than TuranShadow.