Ordering Heuristics for k-clique Listing

Rong-Hua Li[†], Sen Gao[†], Lu Qin[#], Guoren Wang[†], Weihua Yang[‡], Jeffrey Xu Yu[§]

[†]*Beijing Institute of Technology, China;* [#]*University of Technology, Sydney, Australia;*
[‡]*Taiyuan University of Technology, China;* [§]*The Chinese University of Hong Kong, Hong Kong, China.*
rhli@bit.edu.cn; Gawssin@gmail.com; Lu.Qin@uts.edu.au;
wanggrbit@126.com; yangweihua@tyut.edu.cn; yu@se.cuhk.edu.hk

***Abstract*—Listing all k-cliques in a graph is a fundamental graph mining problem that finds many important applications in community detection and social network analysis. Unfortunately, the problem of k-clique listing is often deemed infeasible for a large k, as the number of k-cliques in a graph is exponential in the size k. The state-of-the-art solutions for the problem are based on the ordering heuristics on nodes which can efficiently list all k-cliques in large real-world graphs for a small k (e.g., $k \leq 10$). Even though a variety of heuristic algorithms have been proposed, there still lacks a thorough comparison to cover all the state-of-the-art algorithms and evaluate their performance using diverse real-world graphs. This makes it difficult for a practitioner to select which algorithm should be used for a specific application. Furthermore, existing ordering based algorithms are far from optimal which might explore unpromising search paths in the k-clique listing procedure. To address these issues, we present a comprehensive comparison of all the state-of-the-art k-clique listing/counting algorithms. We also propose a new color ordering heuristics based on greedy graph coloring techniques which is able to significantly prune the unpromising search paths. We compare the performance of 8 various algorithms using 17 large real-world graphs with up to 3 million nodes and 100 million edges. The experimental results reveal the characteristics of different algorithms, based on which we provide useful guidance for selecting appropriate techniques for different applications.**

I. Introduction

Real-world graphs, such as social networks, biological networks, and communication networks often consist of cohesive subgraph structures. Mining cohesive subgraphs from a graph is a fundamental problem in network analysis which has attracted much attention in the database and data mining communities [1]–[6]. Perhaps the most elementary cohesive substructure in a graph is the k-clique structure which has been widely used in a variety of network analysis applications [7]–[12].

Given a graph G, a k-clique is a subgraph with k nodes such that each pair of nodes is connected with an edge. Listing all k-cliques in a graph is a fundamental graph mining operator which finds important applications in community detection and social network analysis. In particular, Palla et al. [7] proposed a k-clique percolation approach to detect overlapping communities in a network, in which a k-clique listing algorithm is used for computing all k-cliques. Mitzenmacher et al. [9] presented a algorithm to find large near cliques which also requires to list all k-cliques. Sariyüce et al. [3] developed a nucleus decomposition method to reveal the hierarchy of dense subgraphs, in which listing all k-cliques is an important building block. Tsourakakis [13] investigated a k-clique densest subgraph problem which also makes use of k-cliques as building blocks. In addition, algorithms for k-clique listing have also been used for story identification in social media [14] and detect the latent higher-order organization in real-world networks [10].

Motivated by the above applications, several practical algorithms have been developed for listing/counting all k-cliques in large real-world graphs [11], [12], [15], [16]. Chiba and Nishizeki [15] developed the first practical algorithm (referred to as the Chiba-Nishizeki algorithm) to solve such a problem, which is turned out to be efficient for real-world graphs. An appealing feature of the Chiba-Nishizeki algorithm is that its running time relies mainly on the arboricity [17] of the graph, which is a sparsity measure of a graph and it is typically very small for real-world graphs [18], [19]. To improve the Chiba-Nishizeki algorithm, Ortmann and Brandes [16] proposed a general ordering-based framework to list triangles in a graph which can also be applied to list k-cliques. Compared to the Chiba-Nishizeki algorithm, the striking feature of the ordering-based framework is that it can be easily parallelized. Danisch et al. [12] also developed a similar ordering-based framework to list all k-cliques, with a particular focus on a degeneracy-ordering based algorithm. They showed that the degeneracy-ordering based algorithm is significantly faster than the Chiba-Nishizeki algorithm. However, they do not compare their algorithm with the other ordering-based algorithms, such as the algorithm based on the degree ordering [16]. To estimate the number of k-cliques, Jain and Seshadhri [11] developed an elegant randomized algorithm called Turán-Shadow based on the classic Turán theorem [20]. The Turán-Shadow algorithm is able to quickly estimate the count of k-cliques, but it cannot output all k-cliques or obtain an exact count.

Although the significance of the k-clique listing problem and the many efforts devoted to investigating it, a comprehensive experimental evaluation of various algorithms for the problem still appears elusive, with existing studies being incomplete by only considering a subset of algorithms (e.g., [12]), or not applying to list general k-cliques (e.g., [16]). This renders it difficult for a practitioner to determine which algorithm should be used for a specific application. Furthermore, existing ordering based algorithms, such as [12] and [16], are far from optimal which may explore many unpromising search paths in the k-clique listing procedure. To address these issues, we carry out an extensive experimental comparison of various algorithms for the k-clique listing problem. We also propose a new color ordering heuristics based on greedy graph coloring techniques [21], [22] which can significantly prune

unpromising search paths. Using a variety of large real-world graphs with up to 3 million nodes and 100 million edge, we compare the performance of 8 different algorithms, including the Chiba-Nishizeki algorithm [15], the degree-ordering based algorithm [16], the degeneracy-ordering based algorithm [12], three proposed color-ordering based algorithms, the optimized degree-ordering based algorithm, and the Turán-Shadow algorithm [11]. Our experimental results reveal the characteristics of different algorithms, based on which we present useful guidance for the selection of appropriate methods for various scenarios.

In summary, the contributions of this paper are:

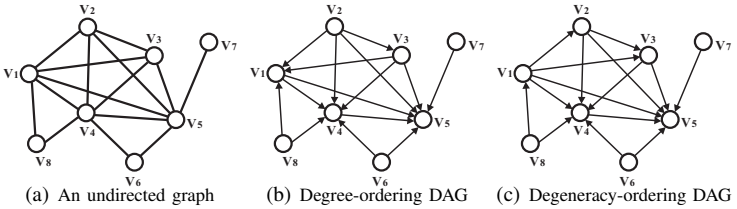
1) We present a thorough experimental study of the known algorithms for listing/counting k-cliques using a variety of large real-world graphs. To the best of our knowledge, this is the first work that compares all the practical k-clique listing/counting techniques from an empirical viewpoint.
2) We propose a new color ordering heuristics, based on which we develop three color-ordering based algorithms for k-clique listing. An appealing feature of the color-ordering based algorithms is that they can significantly prune unpromising search paths in the k-clique listing procedure, which allows us to list k-cliques for a large k value. Moreover, our optimized color-ordering based algorithm can also achieve the same time and space complexity as those of the state-of-the-art algorithm. The experimental results indicate that the opitmized color-ordering based algorithm outperforms all other algorithms for listing k-cliques.
3) We also evaluate the parallel variants of all ordering-based algorithms, and the results show a high degree of parallelism of the ordering-based algorithms.
4) The source codes are publicly accessible at Github[1].

Organization. Section II introduces some useful notations and concepts on the k-clique listing problem. Section III presents our color-ordering based techniques and also reviews all other practical algorithms that we evaluated. The experimental results are reported in Section IV. We review the related work in Section V, and conclude this work in Section VI.

II. Preliminaries

Let $G = (V, E)$ be an undirected graph, where V ($|V| = n$) and E ($|E| = m$) denote the set of nodes and edges respectively. We denote with $N_u(G)$ the set of neighbor nodes of u in G, and $d_u(G) = |N_u(G)|$ denotes the degree of u in G. A subgraph $H = (V_H, E_H)$ is called an induced subgraph of G if $V_H \subseteq V$ and $E_H = \{(u, v)|(u, v) \in E, u \in V_H, v \in V_H\}$. Given a graph G and an integer k, a k-core, denoted by C_k, is a maximal induced subgraph of G such that every node in C_k has a degree no smaller than k, i.e., $d_u(C_k) \geq k$ for every $u \in C_k$ [23]. The core number of a node u, denoted by c_u, is the largest integer k such that there exists a k-core containing u [23]. The maximum core number of a graph G, denoted by δ, is the maximum value of core numbers among all nodes in G. The maximum core number δ is also referred to as the degeneracy of G [19].

[1]https://github.com/Gawssin/kCliqueListing



(a) An undirected graph (b) Degree-ordering DAG (c) Degeneracy-ordering DAG

Fig. 1. Running example

The degeneracy of a graph is closely related to a classic concept called arboricity [17], which is frequently used to measure the sparsity of a graph. Specifically, the arboricity of a graph G, denoted by α, is defined as the minimum number of forests into which its edges can be partitioned. It is well known that the degeneracy of G is a 2-approximation of the arboricity α, i.e., $\alpha \leq \delta \leq 2\alpha - 1$. Note that the core numbers of nodes can be computed in linear time using the classic core-decomposition algorithm [24]. As a result, the degeneracy of a graph can be efficiently determined. The arboricity of a graph, however, is hard to compute [25], therefore practitioners often use the degeneracy to approximate the arboricity of a graph [26].

Another related concept is called h-index of a graph G. The h-index of G is defined as the maximum integer η such that the graph contains η nodes of degree at least η [27]. More formally, $\eta \triangleq \arg\max_k(|\{u|d_u(G) \geq k, u \in V\}| \geq k)$. As shown in [18], η is an upper bound of α, and it is also bounded by $\sqrt{2m}$, i.e., $\alpha \leq \eta \leq \sqrt{2m}$. Notice that the degeneracy, arboricity, and h-index are often very small in real-world graphs [19]. The following example illustrates these concepts.

Example 1. *Consider a graph in Fig. 1(a). We can easily derive that the degeneracy of the graph is 4, since the maximum core number is 4 in this graph. The h-index of the graph is 4, because there are 4 nodes having degrees no less than 4, and there do not exist 5 nodes with degrees no less than 5. Also, it can be shown that the arboricity of the graph is 3, because the graph can be divided into 3 forests and it cannot be decomposed into 2 forests.*

Given a graph G and a parameter k, the k-clique listing/counting problem is a problem of listing/counting all complete subgraphs with size k in G. The state-of-the-art k-clique listing/counting algorithms are based on a graph orientation framework [12], [16], [28]. Let $\pi : V \to \{1, \cdots, n\}$ be a fixed total ordering on nodes in G. Then, for any undirected graph $G = (V, E)$, we are able to obtain a directed graph, denoted by $\vec{G} = (V, \vec{E})$, by orienting each edge $e \in E$ from the lower numbered node to the higher numbered node. More specifically, for each undirected edge $(u, v) \in E$, we obtain a directed edge $(u, v) \in \vec{E}$ if $\pi(u) < \pi(v)$ based on the total order π. Clearly, the directed graph $\vec{G}$ obtained by such an edge-orientation procedure is a directed acyclic graph (DAG), i.e., $\vec{G}$ cannot contain a directed cycle. The algorithms on the basis of such a graph orientation framework

are referred to as ordering-based algorithms. In the following section, we will describe all the practical algorithms for k-clique listing/counting.

III. Algorithms

In this section, we outline five types of practical algorithms that we consider for the k-clique listing/counting problem: (i) the classic Chiba-Nishizeki algorithm [15]; (ii) the degree-ordering based algorithm [16], [28] and the degeneracy-ordering based algorithm [12], which are the two state-of-the-art k-clique listing algorithms; (iii) the color-ordering based algorithms proposed in this work; (iv) the Turán-Shadow algorithm [11], which is the state-of-the-art algorithm for estimating the k-clique counts in a graph. Interested readers are referred to Section V for a summary of other existing algorithms for k-clique listing/counting.

A. The Chiba-Nishizeki Algorithm

We start by describing the classic Chiba-Nishizeki algorithm [15] which is the first practical algorithm for listing k-cliques. The algorithm first sorts the nodes in a non-increasing order of degree (line 7 of Algorithm 1). Then, the algorithm processes the nodes following this order (line 8 of Algorithm 1). For each node v, the algorithm creates a subgraph G_v induced by v's neighbors, and then recursively executes the same procedure on such an induced subgraph (lines 9-10 of Algorithm 1). It should be note that when handling an induced subgraph G_v, the algorithm needs to reorder the nodes in G_v based on their degrees. After processing a node v, the algorithm removes v from the current graph to avoid that any k-clique containing v is repeatedly listed (line 11 of Algorithm 1). Algorithm 1 shows the pseudocode of the Chiba-Nishizeki algorithm.

The striking feature of Algorithm 1 is that its time complexity is closely related to the arboricity of the graph. For many real-world graphs, the arboricity is typically very small, thus the Chiba-Nishizeki algorithm is efficient in practice.

Theorem 1. *([15]) Given a graph G and an integer k, Algorithm 1 lists all k-cliques in $O(km\alpha^{k-2})$ time using linear space, where α is the arboricity of the graph.*

As pointed out in [12], the main defect of Algorithm 1 is that it is difficult to be parallelized. This is because, the algorithm includes a sequential step in line 11, which leads to that the current iteration on the subgraph induced by v_i's neighbors (line 9) depends on the graph $G\backslash\{v_{i-1}\}$ obtained in the previous iteration. Such a sequential step makes an efficient parallelization of the algorithm non-trivial. The ordering-based algorithms discussed in the following section can circumvent this issue.

B. Existing Ordering Based Algorithms

In this subsection, we describe two ordering based algorithms which turn out to be the most efficient algorithms for the k-clique listing problem [12], [16]. The ordering-based algorithms were originally designed to list all triangles in an undirected graph [16]. Recently, it is successfully extended to list all k-cliques in an undirected graph [12], [28]. These algorithms start by computing a total ordering on nodes, and then constructing a DAG based on the total ordering. After that, the algorithms list all k-cliques on the DAG which can prevent that a same k-clique is listed more than once. Specifically, by the ordering based algorithms, each k-clique R is only listed once when the algorithm processes the node in R with the smallest value in the total order. Algorithm 2 shows the framework of the ordering based algorithms.

Algorithm 1: The Chiba-Nishizeki algorithm

```
   Input: An graph G and an integer k
   Output: All k-cliques
1  Arbo (G, ∅, k);
2  Procedure Arbo (G, R, l);
3  if l = 2 then
4      for each edge (u, v) in G do
5          output a k-clique R ∪ {(u, v)};
6  else
7      Sort the nodes in G such that d_{v_1}(G) ≥ ··· ≥ d_{v_{|V_G|}}(G);
8      for i = 1 to |V_G| do
9          Let G_{v_i} be the subgraph of G induced by the set of neighbors of v_i;
10         Arbo (G_{v_i}, R ∪ {v_i}, l − 1);
11         Delete v_i from G;
```

Algorithm 2: The Ordering Based Framework

```
   Input: An graph G and an integer k
   Output: All k-cliques
1  Let π be a total ordering on nodes; /* degree ordering or degeneracy ordering */;
2  Let G⃗ be a DAG generated by π;
3  ListClique(G⃗, ∅, k);
4  Procedure ListClique(G⃗, R, l);
5  if l = 2 then
6      for each edge (u, v) in G⃗ do
7          output a k-clique R ∪ {(u, v)};
8  else
9      for each node v ∈ G⃗ do
10         Let G⃗_v be the subgraph of G⃗ induced by all v's out-going neighbors;
11         ListClique(G⃗_v, R ∪ {v}, l − 1);
```

In Algorithm 2, when handling a node v in the DAG $\vec{G}$, the algorithm recursively lists all k-cliques on the subgraph of $\vec{G}$ induced by v's out-going neighbors (lines 10-11). Unlike the Chiba-Nishizeki algorithm, Algorithm 2 does not need to delete a node from the current graph, thus it allows for an efficient parallel implementation (the **for** loop in lines 9-11 can be easily parallelized). The detailed description of the parallel algorithms is presented in Section III-D.

As indicated in [12], the running time of the ordering based algorithms depends mainly on the orderings on nodes. Unfortunately, finding the best ordering for the k-clique listing algorithms is a NP-hard problem [12], thus we resort to seek good heuristic ordering approaches in practice. Below, we discuss two existing ordering heuristics which are the degree ordering [16] and the degeneracy ordering [12].

The degree-ordering based algorithm. Given a graph G, we can construct a total ordering by sorting the nodes in a non-decreasing of degree (break ties by node identities). We refer to such a total ordering as a degree ordering. Let π_d be a

degree ordering. Clearly, for two nodes with identities u, v, and $u < v$, we have $\pi_d(u) \prec \pi_d(v)$ if $d_u(G) \leq d_v(G)$.

Let $\vec{G}_d$ be a DAG generated by the degree ordering π_d. Specifically, for each undirected edge $(u, v) \in G$ with $\pi_d(u) \prec \pi_d(v)$, we creates a directed edge (u, v) in $\vec{G}_d$. Let $N_u^+(\vec{G}_d)$ and $d_u^+(\vec{G}_d)$ be the set of outgoing neighbors of u in $\vec{G}_d$ and the out-degree of u, respectively. An appealing feature of such a DAG is that the out-degree of a node in $\vec{G}_d$ is bounded by the h-index of G.

Lemma 1. *For any $u \in \vec{G}_d$, $d_u^+(\vec{G}_d) \leq \eta$, where η is the h-index of G.*

Proof. Let $\bar{N}_u = \{v | d_u(G) \leq d_v(G), v \in N_u(G)\}$. To prove the lemma, it is sufficient to show $|\bar{N}_u| \leq \eta$. We can prove this by a contradiction. Suppose to the contrary that $|\bar{N}_u| > \eta$. Then, by assumption, u has more than η neighbors that have degrees larger than $d_u(G)$. Since $d_u(G) \geq |N_u| > \eta$, there are $\eta + 1$ neighbors with degree no less than $\eta + 1$ for node u. As a consequence, the h-index of G must be no less than $\eta + 1$, which is a contradiction. □

Example 2. *Reconsider the graph in Fig. 1(a), we can easily derive that the node ordering $(v_7, v_6, v_8, v_2, v_3, v_1, v_4, v_5)$ is a degree ordering. Based on this ordering, we can obtain a DAG as shown in Fig. 1(b). Clearly, in this DAG, the maximum out-degree is 3, which is smaller than the h-index of the graph ($\eta = 4$).*

Algorithm 2 equipped with a degree ordering heuristics is referred to as a degree-ordering based algorithm. Since the out-degree of the nodes in $\vec{G}_d$ is bounded by the h-index of the original undirected graph G (Lemma 1), we are capable of deriving the time complexity of the degree-ordering based algorithm as follows.

Theorem 2. *Given a graph G and an integer k, the degree-ordering based algorithm lists all k-cliques in $O(km(\eta/2)^{k-2})$ time using linear space, where η is the h-index of G.*

Proof. Let $\vec{G}_d$ be the DAG obtained by the degree ordering, and $N_u^+(\vec{G}_d)$ be the set of outgoing neighbors of a node u in $\vec{G}_d$. By Lemma 1, the out-degree of any node v in $\vec{G}_d$, denoted by $d_v^+(\vec{G}_d)$, is bounded by η. Let $T(l, \vec{G}_d)$ be the running time of the ListClique procedure in Algorithm 2. By the results established in [12], we have $T(2, \vec{G}_d) = O(k|E(\vec{G}_d)|)$, and for $l > 2$, we have

$$T(l, \vec{G}_d) = O(\sum_{u \in V(\vec{G}_d)} \sum_{v \in N_u^+(\vec{G}_d)} d_v^+(\vec{G}_d)) + \sum_{u \in V(\vec{G}_d)} T(l-1, \vec{G}_d[N_u^+(\vec{G}_d)]), \quad (1)$$

where $\vec{G}_d[N_u^+(\vec{G}_d)])$ denotes the subgraph of $\vec{G}_d$ induced by $N_u^+(\vec{G}_d)$. Since $d_v^+(\vec{G}_d) \leq \eta$, $O(\sum_{u \in V(\vec{G}_d)} \sum_{v \in N_u^+(\vec{G}_d)} d_v^+(\vec{G}_d))$ is bounded by $O(\eta |E(\vec{G}_d)|)$. Similarly, we can easily derive that $\sum_{u \in V(\vec{G}_d)} |E(\vec{G}_d[N_u^+(\vec{G}_d)])| \leq |E(\vec{G})| \times \eta/2$. Then, based on a similar inductive argument shown in [12], we are able to obtain that $T(l, \vec{G}_d) \leq c \times (k + l/2) \times (\eta/2)^{l-2} |E(\vec{G}_d)|$, where c is a constant. As a result, the time complexity of Algorithm 2 with degree ordering is $O(km(\eta/2)^{k-2})$. Since Algorithm 2 only needs to store the graph and several linear-size data structures, the space overhead of the algorithm is $O(m + n)$. □

Although the worst-case time complexity of the degree-ordering based algorithm might be higher than that of the Chiba-Nishizeki algorithm, our experimental results show that it significantly outperforms the Chiba-Nishizeki algorithm in real-world graphs. The reason is that the h-index is often very small for real-world graphs as observed in [19].

The degeneracy-ordering based algorithm. Given a graph G with degeneracy δ, a node ordering is called a degeneracy ordering if every node in G has δ or fewer neighbors that come later in the ordering [19]. It is easy to derive that the classic core-decomposition algorithm [24] that repeatedly deletes a node with minimum degree can generate a degeneracy ordering. We refer to such a ordering obtained by the core-decomposition algorithm [24] as a core-based degeneracy ordering, and it will be abbreviated as a degeneracy ordering in the rest of this paper. Obviously, the degeneracy ordering π_δ derived by the core-decomposition algorithm is a total ordering, thus the directed graph induced by π_δ is a DAG. We denote such a DAG by $\vec{G}_\delta$. It is easy to show that $d_u^+(\vec{G}_\delta) \leq \delta$ for any node $u \in \vec{G}_\delta$.

Example 3. *As an illustrative example in Fig. 1(a), we can easily derive that the node ordering $\pi_\delta = (v_7, v_6, v_8, v_1, v_2, v_3, v_4, v_5)$ which is generated by the core-decomposition algorithm is a degeneracy ordering. The DAG generated by π_δ is shown in Fig. 1(c). Obviously, the maximum out-degree of this DAG is 4, which is equal to the degeneracy of the graph ($\delta = 4$).*

Algorithm 2 equipped with a degeneracy ordering (line 1 of Algorithm 2) is referred to as a degeneracy-ordering based algorithm. Danisch et al. [12] shows that such a degeneracy-ordering based algorithm has a slightly lower time complexity than that of the Chiba-Nishizeki algorithm.

Theorem 3. *([12]) Given a graph G and an integer k, Algorithm 2 with a degeneracy ordering heuristics lists all k-cliques in $O(km(\delta/2)^{k-2})$ time using linear space, where δ is the degeneracy of G.*

Recall that $\delta \leq 2\alpha - 1$ where α is the arboricity of G. Thus, we have $O(km(\delta/2)^{k-2}) = O(km(\alpha - 1/2)^{k-2})$, which is slightly smaller than $O(km\alpha^{k-2})$. As shown in [12], the running time of the degeneracy-ordering based algorithm is significantly less than that of the Chiba-Nishizeki algorithm, which is also confirmed in our experiments. However, [12] does not compare the performance between the degree-ordering based algorithm and the degeneracy-ordering based algorithm. Although the worst-case time complexity of the degree-ordering based algorithm is higher than that of the degeneracy-ordering based algorithm, our experimental results

indicate that the running time of these two algorithms is comparable.

C. The Color Ordering Based Algorithms

The main defects of existing ordering based algorithms are twofold: (1) the algorithms are very costly for listing k-cliques with a large k, especially when k is close to the size of a maximum clique (denoted by ω); (2) the algorithms are unable to prune unpromising search paths, in which no k-clique can be found. To overcome these shortcomings, we propose a new ordering heuristics, called color ordering, based on the greedy graph coloring technique [21], [22].

Assume that the graph G can be colored using χ colors. Then, we assign an integer color value taking from $[1, \cdots, \chi]$ to each node in G using any greedy coloring algorithm [21], [22] so that no two adjacent nodes have the same color value. After that, we sort the nodes in a non-increasing order based on their color values (break ties by node identities). We refer to such a total ordering as a color ordering, denoted by π_χ. Let $\vec{G}_\chi$ be the DAG induced by π_χ. Clearly, for each node $v \in \vec{G}_\chi$, the color value of v is no less than the color values of its outgoing neighbors. A striking feature of the color ordering is that the color values of the nodes can be used to prune unpromising search paths in the k-clique listing procedure. Algorithm 3 shows the pseudocode of the color-ordering based algorithm.

In Algorithm 3, the algorithm first invokes a greedy coloring algorithm [21], [22] to obtain a valid coloring for nodes (line 1). Based on the color values, the algorithm constructs a total ordering (line 2) and generates a DAG (line 3). After that, the algorithm recursively lists all k-cliques using a similar procedure as used in Algorithm 2. However, unlike Algorithm 2, Algorithm 3 is able to use the color values to prune unpromising search paths. Specifically, in line 11, when the algorithm explores a node v with a color value (denoted by $\mathsf{color}(v)$) smaller than l, the algorithm can safely prune the search paths rooted at v. The reason is as follows. By the color ordering heuristics, each out-going neighbor of v has a color value strictly smaller than $\mathsf{color}(v)$ (because v's out-going neighbors cannot have the same color value as that of v). Since $\mathsf{color}(v) < l$, all v's out-going neighbors have color values strictly smaller than $l-1$. As a result, v does not have $l-1$ out-going neighbors with different colors, indicating that v cannot be contained in any l-clique. Note that such a pruning strategy not only improves the performance of the algorithm, but it also enables the algorithm to list large k-cliques if the value of k is near to a maximum clique size. The traditional ordering-based algorithms introduced in Section III-B, however, cannot be used to list large k-cliques.

Note that in Algorithm 3, the greedy coloring procedure colors the nodes following a fixed node ordering. When processing a node v, the greedy coloring procedure always selects the minimum color value that has not been used by v's neighbors to color v (lines 17-21). Clearly, various node orderings used in the greedy coloring procedure will generate different color orderings (line 15). Below, we introduce two

Algorithm 3: The Color-ordering Based Algorithm

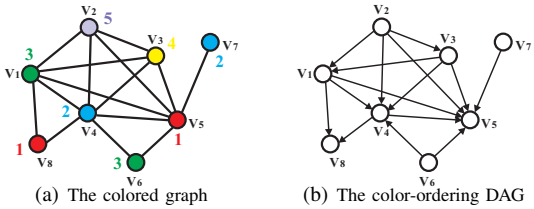
```
   Input: An graph G and an integer k
   Output: All k-cliques
1  color [1, ··· , n] ←GreedyColoring(G);
2  Let π be the total ordering on nodes generated by color values;
3  Let G⃗ be a DAG generated by π;
4  ColorListClique(G⃗, ∅, k);
5  Procedure ColorListClique(G⃗, R, l);
6  if l = 2 then
7      for each edge (u, v) in G⃗ do
8          output a k-clique R ∪ {(u, v)};
9  else
10     for each node v ∈ G⃗ do
11         if color(v) < l then continue;
12         Let G⃗_v be the subgraph of G⃗ induced by all v's out-going neighbors;
13         ColorListClique(G⃗_v, R ∪ {v}, l − 1);
14 Procedure GreedyColoring(G);
15 Let π′ be a total ordering on nodes; /* π′ can be an inverse degree ordering or
       an inverse degeneracy ordering */;
16 flag(i) ← −1 for i = 1, ··· , χ;
17 for each node v ∈ π′ in order do
18     for u ∈ N_v(G) do
19         flag(color(u)) ← v;
20     k ← min{i|i > 0, flag(i) ≠ v};
21     color(v) ← k;
22 return color(v) for all v ∈ G;
```

color orderings based on an *inverse degree ordering* [22] and an *inverse degeneracy ordering* [21].

The degree-based color ordering. This color ordering is generated by the following steps. First, we invoke the greedy coloring procedure to color the nodes following a non-increasing ordering of degree (break ties by node ID). Then, we sorts the nodes in a non-increasing ordering of color value (break ties by node ID). The following example illustrates the detailed procedure for generating such a degree-based color ordering.

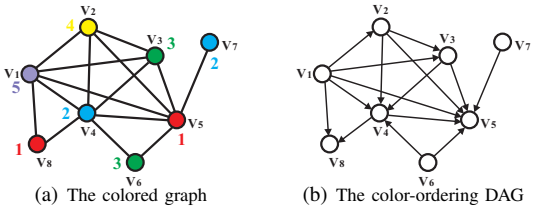
Example 4. *Consider the graph in Fig. 1(a). It is easy to see that $(v_5, v_4, v_1, v_3, v_2, v_8, v_6, v_7)$ is an inverse degree ordering. Following this ordering, the algorithm first colors v_5 with the smallest color 1, and then colors v_4 with a color 2, and the other nodes are iteratively colored in a similar way. Fig.2(a) shows the results of this coloring approach. Based on the color values, we can easily obtain the color ordering $(v_2, v_3, v_6, v_1, v_7, v_4, v_8, v_5)$ (break ties by node ID). The DAG generated by this color ordering is shown in Fig. 2(b).*

Unlike the traditional ordering heuristics shown in Section III-B, the out-degree of a node in the DAG generated by the degree-based color ordering cannot be bounded by the h-index or the degeneracy of the graph. In this case, we can only obtain a trivial upper bound for the out-degree, which is the maximum degree of the graph Δ. As a result, the worst-case time complexity of Algorithm 3 is $O(km(\Delta/2)^{k-2})$ based on a similar analysis in Theorem 2. However, in practice, the running time of such a color-ordering based algorithm can be much lower than the worst-case time complexity as confirmed in our experiments. The reason is as follows. By the *inverse degree ordering*, the node with a high degree might be colored



(a) The colored graph (b) The color-ordering DAG

Fig. 2. Illustration of the degree-based color ordering.



(a) The colored graph (b) The color-ordering DAG

Fig. 3. Illustration of the degeneracy-based color ordering.

with a small color value, thus the high-degree nodes tend to be having a low rank in the degree-based color ordering. As a result, the high-degree nodes might have a relatively small out-degree in the DAG generated by the degree-based color ordering. Based on this, the *real* time consumption of the algorithm can be much lower than the worst-case time complexity $O(km(\Delta/2)^{k-2})$.

The degeneracy-based color ordering. Instead of the *inverse degree ordering*, we can also use an *inverse degeneracy ordering* [21] to color the nodes in Algorithm 3. Note that such an inverse degeneracy ordering can be easily obtained by reversing the node-deletion ordering of the core-decomposition algorithm [24]. The color ordering generated by this approach is referred to as a degeneracy-based color ordering. We use the following example to illustrate such a greedy coloring procedure, as well as the degeneracy-based color ordering.

Example 5. *Reconsider the graph in Fig. 1(a). The ordering* $(v_5, v_4, v_3, v_2, v_1, v_8, v_6, v_7)$ *is an inverse degeneracy ordering. Following this ordering, we can obtain a colored graph shown in Fig. 3(a) by the greedy coloring procedure. Based on the color values, it is easy to derive that* $(v_1, v_2, v_3, v_6, v_4, v_7, v_8, v_5)$ *is a color ordering. Fig. 3(b) shows the DAG generated by this color ordering.*

Similar to the degree-based color ordering, the maximum out-degree of the DAG generated by the degeneracy-based color ordering could be equal to Δ in the worst case. Thus, the worst-case time complexity of Algorithm 3 with a degeneracy-based color ordering is $O(km(\Delta/2)^{k-2})$. However, in practice, the running time of our algorithm is much lower than the worst-case time complexity as verified in our experiments. The reasons are as follows. By the *inverse degeneracy ordering*, a node with a large core number may be assigned by a small color value, thus such a node might have a low rank in the degeneracy-based color ordering. As a result, the DAG generated by the degeneracy-based color ordering might be *similar* to the DAG induced by the degeneracy ordering (see Fig. 1(c) and Fig. 3(b) for example), indicating that the *real* time cost of Algorithm 3 can be much lower than $O(km(\Delta/2)^{k-2})$. In addition, Algorithm 3 also applies the color values to prune unpromising search paths, thus it can be very efficient in practice.

The optimized color-ordering based algorithm. Here we present a simple but effective strategy to reduce the worst-case time complexity of our color-ordering based algorithms. In particular, we can first generate a DAG $\vec{G}$ based on the degeneracy ordering. Let $N_v^+(\vec{G})$ be the set of out-going neighbors of v in $\vec{G}$. Then, for each v, we construct a subgraph $G_v = (V_v, E_v)$ of the original undirected G that is induced by the nodes in $N_v^+(\vec{G})$ ($V_v = N_v^+(\vec{G})$). After that, for each v with $|V_v| \geq k-1$, we iteratively list all $k-1$ cliques in G_v using Algorithm 3. The pseudocode of such an optimized algorithm is shown in Algorithm 4. An important feature of this algorithm is that its worst-case time complexity is $O(km\delta^{k-2})$, which is the same as that of the degeneracy-ordering based algorithm.

Algorithm 4: The Optimized Color-ordering Based Algorithm

Input: An graph G and an integer k
Output: All k-cliques
1 Let π be a degeneracy ordering;
2 Let $\vec{G}$ be a DAG generated by π;
3 Let $N_v^+(\vec{G})$ be the set of out-going neighbors of v;
4 Let $G_v = (V_v, E_v)$ be the subgraph of G induced by the nodes in $N_v^+(\vec{G})$;
5 **for** *each* $v \in G$ **do**
6 **if** $|V_v| \geq k-1$ **then**
7 Invoke Algorithm 3 on the subgraph G_v with parameter $k-1$;

Theorem 4. *Given a graph G and an integer k, Algorithm 4 lists all k-cliques in $O(km(\delta/2)^{k-2})$ time using linear space, where δ is the degeneracy of G.*

Proof. First, based on the degeneracy-ordering DAG (line 2 of Algorithm 4), the out-degree of each node v is bounded by δ. Thus, for each v, $G_v = (V_v, E_v)$ contains at most δ nodes. By the color-ordering based algorithm, we can list all $k-1$ cliques in G_v in $O((k-1)|E_v|(\delta/2)^{k-3})$ time. As a result, the total time complexity of Algorithm 4 is $O(k(\delta/2)^{k-3}\sum_{v\in V}|E_v|)$. Since $\sum_{v\in V}|E_v| \leq \delta/2 \times \sum_{v\in V}|V_v| = \delta/2 \times m$, the worst-case time complexity of Algorithm 4 is bounded by $O(km(\delta/2)^{k-2})$. Since Algorithm 4 only needs to store the graph and several linear-size data structures, the space complexity of the algorithm is linear with respect to (w.r.t.) the graph size. □

Discussions. The above optimization strategy can also be used to reduce the worst-case time complexity of the degree-ordering based algorithm. Specifically, instead of using Algorithm 3, we make use of the degree-ordering based algorithm to list all $(k-1)$-cliques on G_v in the line 7 of Algorithm 4. By a similar analysis in Theorem 4, the time complexity of such an optimized degree-ordering based algorithm is $O(km(\delta/2)^{k-2})$.

D. Parallel Ordering Based Algorithms

In [12], Danisch et al. proposed two general parallel strategies for ordering-based algorithms, namely, NodeParallel and EdgeParallel respectively. Note that all the above ordering-based algorithms can be parallelized using the NodeParallel or the EdgeParallel strategy. Let $\vec{G}_v$ be the subgraph of $\vec{G}$ induced by the outgoing neighbors of v. Recall that by the ordering-based algorithm, the subgraph $\vec{G}_v$ for each node v can be processed independently (see line 9 of Algorithm 2, line 10 of Algorithm 3, and line 5 of Algorithm 4). The NodeParallel strategy processes all of such G_v's in parallel. However, since the k-cliques may not be distributed uniformly in all G_v's, which gives rise to unbalanced workloads on different CPUs. This shortcoming can be alleviated by the EdgeParallel strategy. Let (u, v) be an edge in G, and $\vec{G}_{uv}$ be the subgraph of $\vec{G}$ induced by the common outgoing neighbors of u and v (the subgraph induced by $N_u^+(\vec{G}) \cap N_v^+(\vec{G})$). The EdgeParallel strategy processes all $\vec{G}_{uv}$'s in parallel. Specifically, EdgeParallel invokes the ListClique($\vec{G}_{uv}, \{u, v\}, k-2$) procedure (or the ColorListClique($\vec{G}_{uv}, \{u, v\}, k-2$) procedure for color-ordering based algorithm) for all $\vec{G}_{uv}$'s in parallel. Since the $\vec{G}_{uv}$'s are generally smaller than the $\vec{G}_v$'s, thus EdgeParallel can achieve a higher degree of parallelism, which is also confirmed in our experiments.

Remark. It is worth remarking that the EdgeParallel strategy can also be used for parallelizing Algorithm 4. In particular, for each directed edge (u, v) in the degeneracy-ordering DAG $\vec{G}$, we can obtain a set of common out-going neighbors $N_{uv}^+ = N_u^+(\vec{G}) \cap N_v^+(\vec{G})$. Then, we construct a subgraph G_{uv} of G that is induced by the nodes in N_{uv}^+. After that, the algorithm can process all G_{uv}'s in parallel.

E. The Turán-Shadow Algorithm

The Turán-Shadow algorithm [11] is a randomized algorithm that is designed to estimate the number of k-cliques in an undirected graph. The algorithm involves two sub-procedures: ShadowConstruction and Sampling (Algorithm 5). In the ShadowConstruction procedure, the algorithm constructs a data structure called Turán Shadow based on the classic Turán theorem [20]. Specifically, the Turán theorem states that for any graph G, if the density of G, denoted by $\rho(G) = m/\binom{|n|}{2}$, is larger than $1 - 1/(k-1)$, then G contains a k-clique [20].

For a given integer k, the Turán Shadow $\mathcal{S}$ consists of a set of pairs (H, l), where $H \subseteq V$ is a subset of nodes and $l \leq k$ is an integer. For each pair $(H, l) \in \mathcal{S}$, the density of the subgraph induced by H, denoted by $\rho(G_H)$, is larger than the so-called Turán threshold $1 - 1/(l-1)$. Therefore, for a pair (H, l), the subgraph G_H must contain a l-clique by Turán theorem.

Jain and Seshadhri [11] proposed an elegant refinement procedure to construct such a Turán Shadow. The pseudocode of the refinement procedure is shown in Algorithm 5 (lines 1-12). Initially, the algorithm sets $\mathcal{T} = \{(V, k)\}$ and the Turán Shadow $\mathcal{S} = \emptyset$ (line 2). Then, the algorithm iteratively picks a pair (H, l) from $\mathcal{T}$ that does not satisfy the Turán threshold (line 3). With such a pair (H, l), the algorithm constructs a DAG $\vec{G}_H$ for H based on the degeneracy ordering (lines 4-5). For each node $v \in H$, the algorithm creates an outgoing neighborhood $N_v^+(\vec{G}_H)$ in $\vec{G}_H$ (line 6). Subsequently, the algorithm constructs a set of $|H|$ pairs $\{(N_v^+(\vec{G}_H), l-1) | v \in H\}$. For any pair $(N_v^+(\vec{G}_H), l-1)$ that meets the Turán threshold will go to the Turán Shadow $\mathcal{S}$, otherwise it goes to $\mathcal{T}$ (lines 7-11). After that, the algorithm deletes the pair (H, l) from $\mathcal{T}$ (line 12), and recurses on the updated $\mathcal{T}$ (line 3). The idea behind the ShadowConstruction procedure is that it iteratively refines the pairs in $\mathcal{T}$ until all pairs satisfies the Turán threshold.

Based on the Turán Shadow $\mathcal{S}$, Jain and Seshadhri [11] proved that there exists a one-to-one mapping between a k-clique in G and a l-clique in G_H for a pair $(H, l) \in \mathcal{S}$, where G_H is a subgraph induced by H. As a result, counting the number of k-cliques in G is equivalent to compute the total number of l-cliques in G_H for each pair (H, l). To do this, a simple weighted sampling procedure is sufficient to estimate the l-clique counts in $\mathcal{S}$ [11]. The pseudocode of such a sampling procedure is detailed in Algorithm 5 (lines 13-21). The accuracy and complexity of Algorithm 5 is shown in the following theorem.

Theorem 5. *[11] Given a graph G and an integer k, Algorithm 5 with an appropriate parameter t can output an estimate $\hat{C}_k$ such that $|\hat{C}_k - C_k| \leq \epsilon C_k$ with probability $1 - \tau$, where C_k is the number of k-cliques in G, ϵ and τ are two small constants. The time complexity of Algorithm 5 is $O(|\mathcal{S}| + (\log(\frac{1}{\tau})/\epsilon^2))$, where $|\mathcal{S}|$ is bounded by $O(n\alpha^{k-2})$.*

Although Algorithm 5 has a high time complexity in the worst case, it is turned out to be very efficient in practice, since the size of the Turán Shadow, i.e., $|\mathcal{S}|$, is often not very large for real-world graphs [11].

IV. Experiments

A. Experimental setup

Datasets. We collect 17 large real-world graphs obtained from (http://networkrepository.com/). We divide the datasets into two groups based on the size of a maximum clique (ω): small-ω graphs (the first 9 graphs in Table I), for which we are capable of listing all k-cliques (for all k), and large-ω graphs (the last 8 graphs in Table I), for which we can only list k-cliques for small k values (or large k-cliques with k values near to ω). This is because if ω is large, the number of k-cliques in the maximum clique is exponential large for a relatively large k, and thus any exact k-clique listing/counting algorithm is doomed to failure. For example, if $\omega = 60$ and $k = 20$, then a 60-clique contains 4.2×10^{15} 20-cliques. For each graph dataset, we report its maximum clique size ω, the number of maximum cliques $N_{\max}$, the maximum k-core number δ, and the maximum degree Δ, which could affect the running time of the k-clique listing/counting algorithms. Table I summarizes the statistics of our datasets.

TABLE III
RUNNING TIME OF DIFFERENT ALGORITHMS FOR LISTING ALL k-CLIQUES ($k \geq 3$, IN SECOND)

Dataset	#Cliques	Arbo	Degree	Degen	DegCol	DegenCol	DDegCol	DDegree
Nasasrb	50,915,452,049	INF	9,872	9,965	1,346	1,464	**1,307**	1,950
FBWosn	87,432,996,809	INF	INF	INF	3,171	2,751	**2,119**	3,408
WikiTrust	12,652,027,321	11,568	2,962	2,710	421	430	**326**	503
Youtube	44,272,612	65	20	19	**12**	**12**	17	14
Pokec	3,229,825,345	3,252	1,107	1,086	**236**	241	434	392
WikiCN	17,495,574,003	INF	4,636	4,566	**519**	526	598	790
Shipsec5	12,961,780,899	7,347	2,734	2,435	354	**337**	352	515
BaiduBK	7,968,788,787	6,559	2,167	2,107	**390**	398	492	543
SocFBa	13,238,147,662	13,899	3,522	3,544	786	773	**695**	809

Algorithm 5: The Turán-Shadow Algorithm

Input: An graph G and an integer k
Output: An estimation of the number k-cliques

Procedure ShadowConstruction(G, k);
  $\mathcal{T} \leftarrow \{(V, k)\}$, $\mathcal{S} \leftarrow \emptyset$;
  while $\exists (H, l) \in \mathcal{T}$ *s.t.* $\rho(H) \leq 1 - 1/(l-1)$ **do**
    Let G_H be the subgraph of G induced by H;
    $\vec{G}_H \leftarrow$ construct a DAG by the degeneracy ordering on G_H;
    Let $N_v^+(\vec{G}_H)$ be the set of out-going neighbors of v in $\vec{G}_H$;
    for *each* $u \in H$ **do**
      if $l \leq 2$ *or* $\rho(N_v^+(G_H)) > 1 - 1/(l-2)$ **then**
        $\mathcal{S} \leftarrow \mathcal{S} \cup \{(N_v^+(G_H), l-1)\}$;
      else
        $\mathcal{T} \leftarrow \mathcal{T} \cup \{(N_v^+(G_H), l-1)\}$;
    $\mathcal{T} \leftarrow \mathcal{T} \setminus \{(H, l)\}$;

Procedure Sampling($\mathcal{S}$);
  $w(H) \leftarrow \binom{|H|}{l}$ for each $(H, l) \in \mathcal{S}$;
  $p(H) \leftarrow w_H / \sum_{(H,l)\in\mathcal{S}} w(H)$;
  for $r = 1$ **to** t **do**
    Independently sample (H, l) from $\mathcal{S}$ based on the probability $p(H)$;
    $R \leftarrow$ randomly picking l nodes from H;
    if R forms a l-clique **then** $X_r \leftarrow 1$;
    else $X_r \leftarrow 0$;
  return $\frac{\sum_r X_r}{t} \sum_{(H,l)\in\mathcal{S}} w(H)$;

TABLE I
DATASETS (ω DENOTES THE SIZE OF A MAXIMUM CLIQUE, $N_{\max}$ DENOTES THE NUMBER OF MAXIMUM CLIQUES, δ DENOTES THE MAXIMUM k-CORE NUMBER, AND Δ IS THE MAXIMUM DEGREE; 1K=1,000)

Dataset	$n = \|V\|$	$m = \|E\|$	ω	$N_{\max}$	δ	Δ
Nasasrb	54,870	1,311,227	24	1,939	36	275
FBWosn	63,731	817,090	30	2	85	2K
WikiTrust	138,587	715,883	25	54	65	12K
Youtube	1,157,828	2,987,624	17	2	50	28.8K
Pokec	1,632,803	22,301,964	29	6	48	15K
WikiCN	1,930,270	8,956,902	33	2	128	30K
Shipsec5	179,104	2,200,076	24	744	30	75
BaiduBK	2,140,198	17,014,946	31	4	83	98K
SocFBa	3,097,165	23,667,394	25	35	75	5K
WebSK	121,422	334,419	82	3	82	590
Citeseer	227,320	814,134	87	1	87	1K
WebStan	281,904	1,992,636	61	10	87	39K
DBLP	317,080	1,049,866	114	1	114	343
Digg	770,799	5,907,132	50	192	237	17.6K
Orkut	2,997,166	106,349,209	47	7	254	27.5K
Skitter	1,696,415	11,095,298	67	4	112	35K
DielFilter	420,408	16,232,900	45	15,446	57	302

Algorithms. We evaluate 8 various algorithms in our experiments. Table II shows the details of these algorithms. In particular, Arbo denotes the classic Chiba-Nishizeki algorithm [15] (Algorithm 1). We use a C++ implement of this algorithm provided in [12]. Degree denotes the degree-ordering based algorithm, i.e., Algorithm 2 with a degree ordering. Degen is the degeneracy-ordering based algorithm [12], i.e., Algorithm 2 with a degeneracy ordering. For Degen, we make use of a C++ implementation provided in [12]. DegCol and DegenCol denote our color-ordering based algorithms (Algorithm 3) that use an *inverse degree ordering* and an *inverse degeneracy ordering* for coloring, respectively. DDegCol is an optimized version of DegCol (Algorithm 4). Note that we do not evaluate the optimized version of DegenCol, because the performance of such an optimized algorithm is very similar to that of DDegCol. DDegree is an optimized version of Degree, i.e., DDegree is Algorithm 4 that uses Degree to compute the $(k-1)$-cliques on each G_v (see line 7 of Algorithm 4). TuranSD is the Turán-Shadow algorithm [11] for estimating the k-clique count (Algorithm 5). For TuranSD, we use the C++ implementation provided in [11]. We implement the five algorithms Degree, DegCol, DegenCol, DDegCol, and DDegree in C++. In addition, we also implement the parallel variants of six ordering-based algorithms in C++ and OpenMP.

TABLE II
SUMMARY OF k-CLIQUE LISTING/COUNTING ALGORITHMS

Algorithms	Ordering	Time Complexity	Type	Reference
Arbo	$\times$	$O(km\alpha^{k-2})$	exact	[15]
Degree	degree ordering	$O(km(\eta/2)^{k-2})$	exact	[16], [28]
Degen	degeneracy ordering	$O(km(\delta/2)^{k-2})$	exact	[12]
DegCol	color ordering	$O(km(\Delta/2)^{k-2})$	exact	this paper
DegenCol	color ordering	$O(km(\Delta/2)^{k-2})$	exact	this paper
DDegCol	color ordering	$O(km(\delta/2)^{k-2})$	exact	this paper
DDegree	degree ordering	$O(km(\delta/2)^{k-2})$	exact	this paper
TuranSD	$\times$	$O(n\alpha^{k-1})$	approx.	[11]

Experimental settings. We conduct our experiments on a Linux machine equipped with 2 Intel Xeon 2.40GHz CPUs with 12 cores (a total of 24 threads) and with 128 GB RAM. Unless otherwise specified, we evaluate all algorithms with a varying k from 3 to 9. We also evaluate three color-ordering based algorithms and the DDegree algorithm by varying k from $\omega-8$ to ω, where ω is the size of a maximum clique. Note that except the color-ordering based algorithms and DDegree, all the other exact algorithms presented in Section III are intractable for listing large k-cliques when k is varied from $\omega-8$ to ω. In addition, we also evaluate the parallel implementations of all the ordering-based algorithms by varying the number of threads from 1 to 24. In all our experiments, we set the time limit to 8 hours for each algorithm. The running time of any algorithm that exceeds 8 hours is recorded with a special symbol "INF".

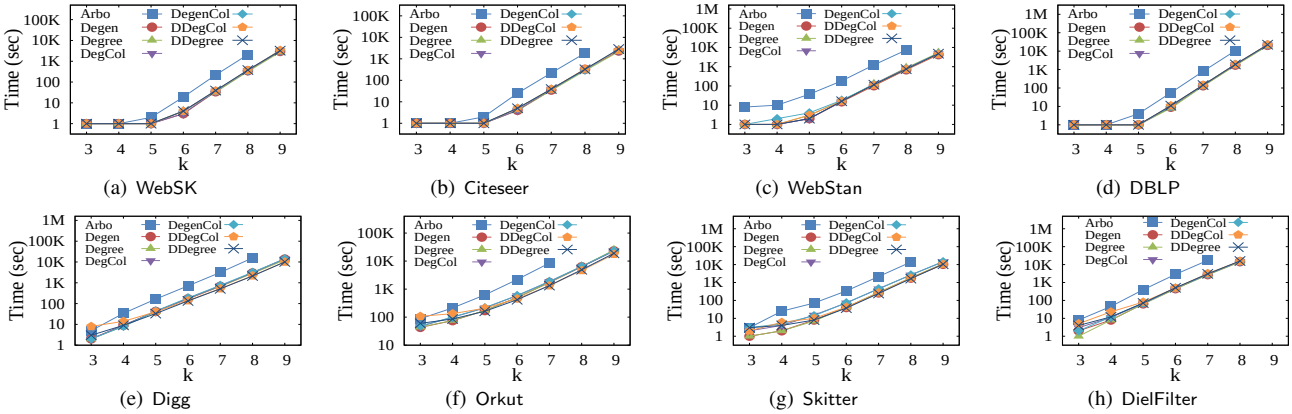


Fig. 5. Running time of different algorithms on large-ω graphs with varying k from 3 to 9

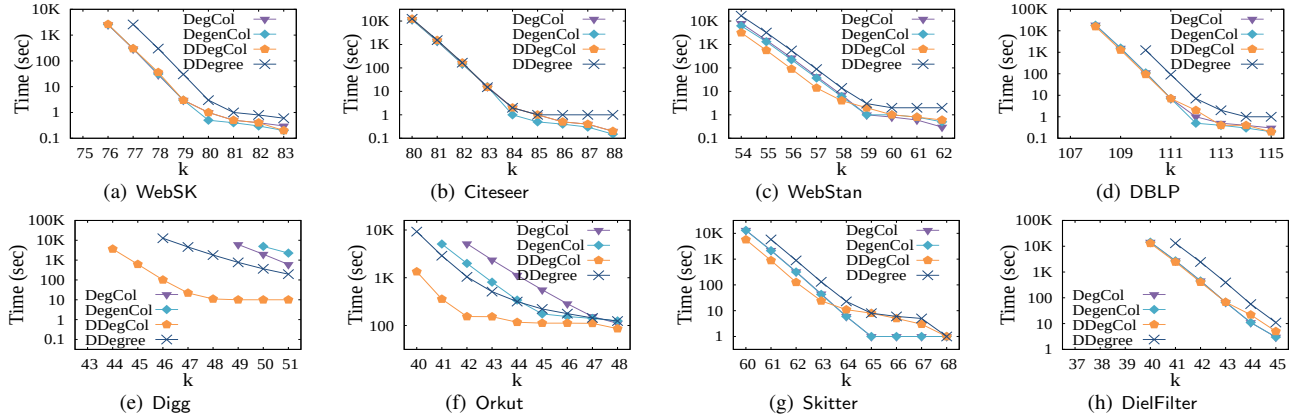


Fig. 6. Running time of different algorithms on large-ω graphs with varying k from $\omega - 8$ to ω

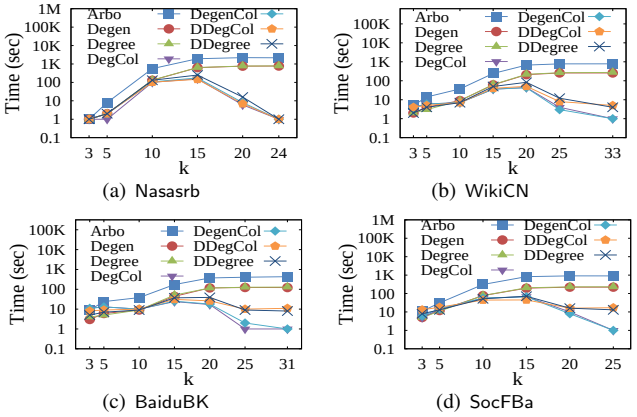


Fig. 4. Running time of various algorithms on small-ω graphs (vary k)

B. Results on Small-ω Graphs

Running time for listing all k-cliques. Table III reports the running time of various exact algorithms for listing all k-cliques (k is varied from 3 to ω) on 9 small-ω graphs. On these datasets, we observe that the three color-ordering based algorithms (DegCol, DegenCol, and DDegCol) achieve similar performance, and all of them significantly outperform the other algorithms. For example, in the Nasasrb dataset, the running time of DegCol, DegenCol, and DDegCol are 1,346, 1,464, and 1,307 seconds respectively. However, in the same dataset, the running time of DDegree, Degree and Degen are 1,950, 9,872, and 9,965 seconds respectively. Arbo performs even worse and it is intractable for listing all k-cliques in this dataset (i.e., it cannot finish in 8 hours). This is because all our color-ordering based algorithms are equipped with a color-value based pruning strategy, which can largely prune the search paths when listing large k-cliques. In general, the traditional ordering-based algorithms (Degree and Degen) are significantly faster than the classic Chiba-Nishizeki algorithm (Arbo), which are consistent with the results shown in [12], [16]. In addition, we can see that the optimized degree-ordering based algorithm (DDegree) performs very well. The running time of DDegree is slightly higher than those of the color-ordering based algorithms, but it is considerably lower than that of the traditional ordering based algorithms (Degree and Degen). The reason could be that DDegree applies the out-degrees to prune the nodes that are definitely not contained in any k-clique (see line 6 of Algorithm 4); such a pruning rule might be very effective when listing large k-cliques.

Running time of various algorithms with varying k. We plot the running time achieved by each algorithm as a function of k on Nasasrb, WikiCN, BaiduBK, and SocFBa in Fig. 4. The results on the other small-ω graphs are consistent. As can be seen, the running time of Arbo, Degree, and Degen increases as k increases. However, for the three color-ordering

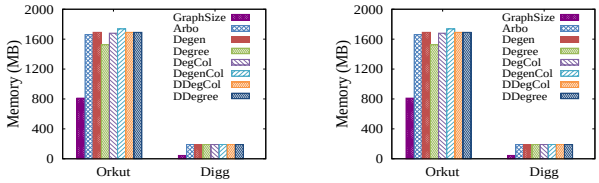
based algorithms and the DDegree algorithm, the running time first increases as k increases to $\omega/2$, and then drops when k increases to ω. The reason could be that the pruning performance of these four algorithms becomes more effective for a larger k. The results also suggest that the performance of the color-ordering based algorithms and DDegree seems to match the number of outputs of the problem, since the number of k-cliques of a graph also exhibits a similar function of k. In addition, we observe that the three color-ordering based algorithms are slightly faster than DDegree, and all of them outperform the other competitors. We also note that both Degree and Degen significantly outperform Arbo, which are consistent with the previous results. In summary, for small-ω graphs (e.g., $\omega \leq 30$), our color-ordering based algorithms are the winners, thus they are recommended to use in this case.

C. Results on Large-ω Graphs

Fig. 5 shows the running time of various exact algorithms on 8 large-ω graphs, with varying k from 3 to 9. As desired, the running time of each algorithm increases with an increasing k. All the ordering-based algorithms achieve similar performance, and they are significantly faster than the Arbo algorithm. In addition, we can also observe that both the optimized color-ordering based algorithm (DDegCol) and the optimized degree-ordering based algorithm (DDegree) seem to be slightly faster than the other ordering-based algorithms (especially for a large k). Taking the Orkut dataset as an example (Fig. 5(f)), DDegCol and DDegree consumes $17,595$ and $18,902$ seconds respectively when $k = 9$. However, under the same parameter setting, Degree, Degen, DegCol, and DegenCol takes $23,200$, $24,827$, $22,610$, $23,467$ seconds respectively. The DDegCol algorithm, for example, improves the running time over Degree, Degen, DegCol, and DegenCol by 24%, 29%, 25%, and 22%, respectively. The reason could be the pruning rule equipped within the optimized algorithms (Algorithm 4) is effective for a relatively large value of k.

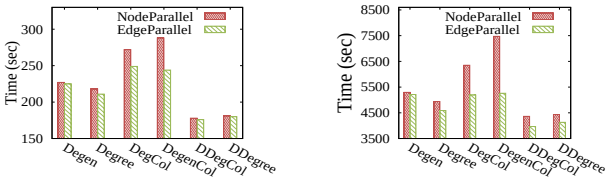
We also plot the running time of DegCol, DegenCol, DDegCol, and DDegree on large-ω graphs in Fig. 6, with varying k from $\omega - 8$ to ω. Note that the Arbo algorithm and the traditional ordering-based algorithms (Degree and Degen) are intractable for listing k-cliques if k is near to the maximum clique size. We observe that the performance of the three color-ordering based algorithms are significantly better than DDegree on most datasets, due to the powerful color-value based pruning technique. As two exceptions, on Digg and Orkut, the DDegree algorithm is faster than DegCol and DegenCol, but it is considerably worse than DDegCol. The reason could be that both the degeneracy and the maximum degree of these two graphs are very large, thus the number of colors obtained by the greedy coloring procedure in both DegCol and DegenCol can also be very large, which reduces the color-value based pruning performance in Algorithm 3.

In summary, all the ordering-based algorithms perform very well for listing small k-cliques on large-ω graphs. The Arbo algorithm and the traditional ordering-based algorithms cannot be used to list k-cliques when k is near to the maximum

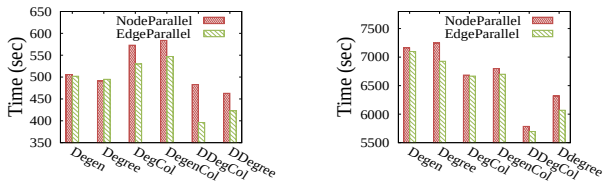


(a) $k = 6$ (b) $k = 8$

Fig. 7. Memory overheads of different algorithms

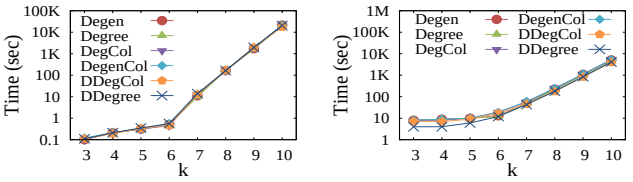


(a) Digg ($k = 8$) (b) Digg ($k = 10$)

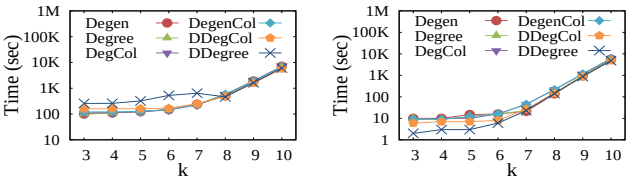


(c) Orkut ($k = 8$) (d) Orkut ($k = 10$)

Fig. 8. NodeParallel vs. EdgeParallel strategies



(a) DBLP (b) Digg



(c) Orkut (d) Skitter

Fig. 9. Running time of various algorithms with the EdgeParallel strategy

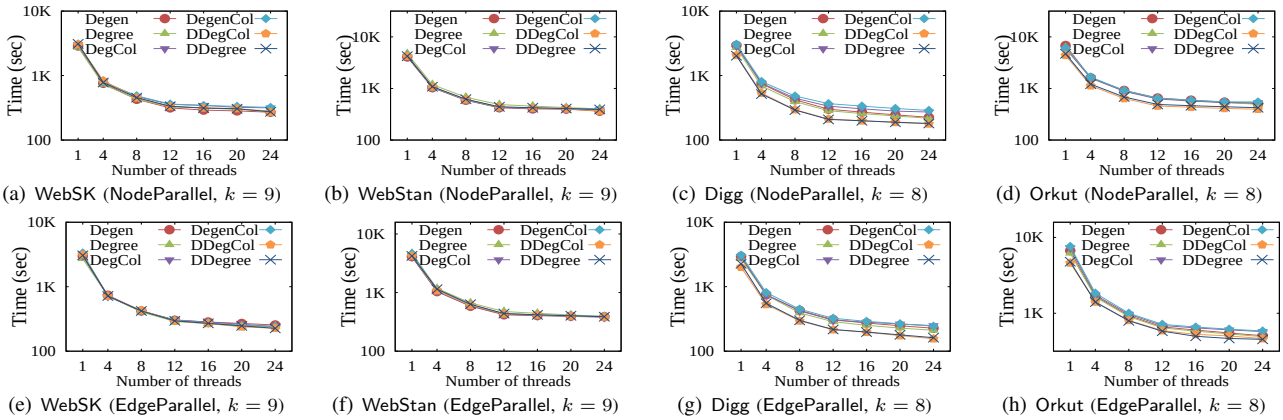
clique size. For both small and large values of k, the overall performance of the optimized color-ordering based algorithm (DDegCol) seems to be better than the other algorithms. Hence, such an optimized color-ordering based algorithm is recommended to use for large-ω graphs.

D. Memory Overheads

We evaluate the memory overheads of various exact algorithms on Digg and Orkut for $k = 6$ and $k = 8$ respectively. The results are shown in Fig. 7. Similar results can also be observed on the other datasets and for the other values of k. As desired, the space usage of each algorithm is only a few times larger than the graph size, since all the algorithms presented in Section III have linear space complexity. These results indicate that all the exact algorithms are space efficient for listing k-cliques in large real-world graphs.

E. Evaluation of Parallel Algorithms

In this subsection, we carry out a set of experiments to evaluate the performance of six ordering-based parallel algorithms.



(a) WebSK (NodeParallel, $k = 9$) (b) WebStan (NodeParallel, $k = 9$) (c) Digg (NodeParallel, $k = 8$) (d) Orkut (NodeParallel, $k = 8$)

(e) WebSK (EdgeParallel, $k = 9$) (f) WebStan (EdgeParallel, $k = 9$) (g) Digg (EdgeParallel, $k = 8$) (h) Orkut (EdgeParallel, $k = 8$)

Fig. 10. Speedup of different parallel algorithms

Note that the Arbo algorithm cannot be parallelized, thus we preclude it in these experiments.

NodeParallel **vs.** EdgeParallel **strategies.** We start by comparing the performance of the NodeParallel algorithms and the EdgeParallel algorithms. Fig. 8 shows the results on Digg and Orkut for $k = 8$ and $k = 10$ respectively. Note that for all parallel algorithms, we are able to list all 10-cliques on all datasets in 8 hours. Again, the results on the other datasets and for the other values of k are consistent. From Fig. 8, we can see that the performance of all algorithms with the NodeParallel strategy is worse than that with the EdgeParallel strategy. This is because for all algorithms, the EdgeParallel strategy achieves a relatively larger degree of parallelism than the NodeParallel strategy. Additionally, we observe that DDegCol with the EdgeParallel strategy significantly outperforms the other parallel algorithms, which further demonstrates the superiority of the DDegCol algorithm.

Evaluation of EdgeParallel **algorithms with varying** k**.** Fig. 9 plots the running time of different EdgeParallel algorithms as a function of k on the DBLP, Digg, Orkut, and Skitter datasets. Similar results can be obtained on the other datasets. As desired, the running time of each parallel algorithm increases as k increases. All algorithms achieve similar running times. For a large value of k, both the parallel DDegCol and DDegree algorithms slightly outperform the other competitors. For example, in Skitter, the running time of the parallel DDegCol and DDegree algorithms are 4,834 and 4,873 seconds respectively. In contrast, the running time achieved by the parallel Degree, Degen, DegCol, and DegenCol algorithms are slightly higher, which are 5,672, 5,155, 5,989, and 5,983 seconds, respectively. The results are consistent with what we observe in Fig. 8.

Speedup of different parallel algorithms. We evaluate the speedup of a variety of parallel algorithms, in which the speedup is defined as the running time of the sequential algorithm divided by the running time of the parallel algorithms when using t threads (t is varied from 1 to 24). Figs. 10(a-d) and Figs. 10(e-h) show the speedup of different NodeParallel and EdgeParallel algorithms, respectively. We can see that the running time of the EdgeParallel algorithms drops more quickly than those of the NodeParallel algorithms on the same dataset, indicating that the EdgeParallel strategy is better to balance the computational loads across the threads than the NodeParallel strategy. We note that the speedup of the NodeParallel algorithms becomes worse when the number of threads is larger than 12, which suggests that the NodeParallel strategy might be inappropriate for listing k-cliques in a massively parallel setting. In addition, we can also observe that both the parallel DDegCol and DDegree algorithms outperform the other parallel algorithms, which are consistent with our previous results.

In summary, the EdgeParallel strategy is significantly better than the NodeParallel strategy for all ordering-based algorithms. Generally, the former can achieve a good speedup on most datasets, while the latter is inappropriate for listing k-cliques in the massively parallel setting. The parallel variants of DDegCol and DDegree are slightly faster than the other algorithms, thus we recommend to use these parallel k-clique listing algorithms in practice.

F. Results of the Turán-Shadow Algorithm

In this subsection, we present our evaluation of the Turán-Shadow algorithm (TuranSD) in terms of the relative errors (RE), running time, and space usage. The relative error is defined as $|N_k - \hat{N}_k|/N_k$, where N_k is the *true* k-clique count and $\hat{N}_k$ is an estimating count. To compute the relative error, we run the sampling procedure in the TuranSD algorithm (Algorithm 5) 100 times (each time drawing 50,000 samples as suggested in [11], i.e., $t = 50,000$ in Algorithm 5), and then take the average count over the 100 runs as the final estimating count. Note that the Turán-Shadow construction procedure in TuranSD is only executed once.

Relative error of TuranSD**.** Fig. 11 shows the relative error of TuranSD on four small-ω graphs with varying k from 3 to ω. Similar results can also be observed on the other small-ω graphs. From Fig. 11, we can see that the TuranSD algorithm is highly accurate in most cases, which has relative error lower than 1% for almost all k values on all datasets. As an exception, when $k = \omega$, the relative error of TuranSD

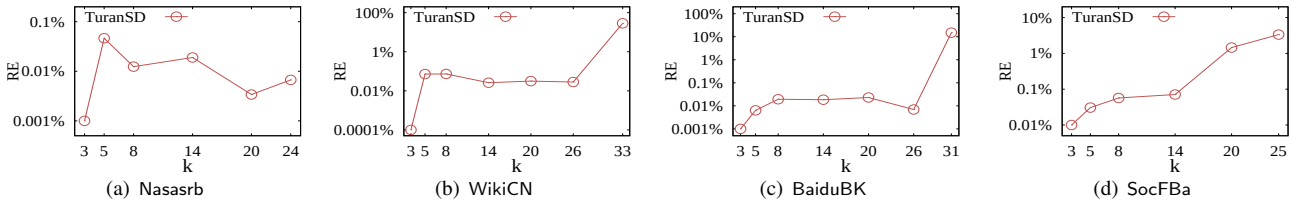


Fig. 11. Relative error (RE) of the Turán-Shadow algorithm on small-ω graphs (vary k)

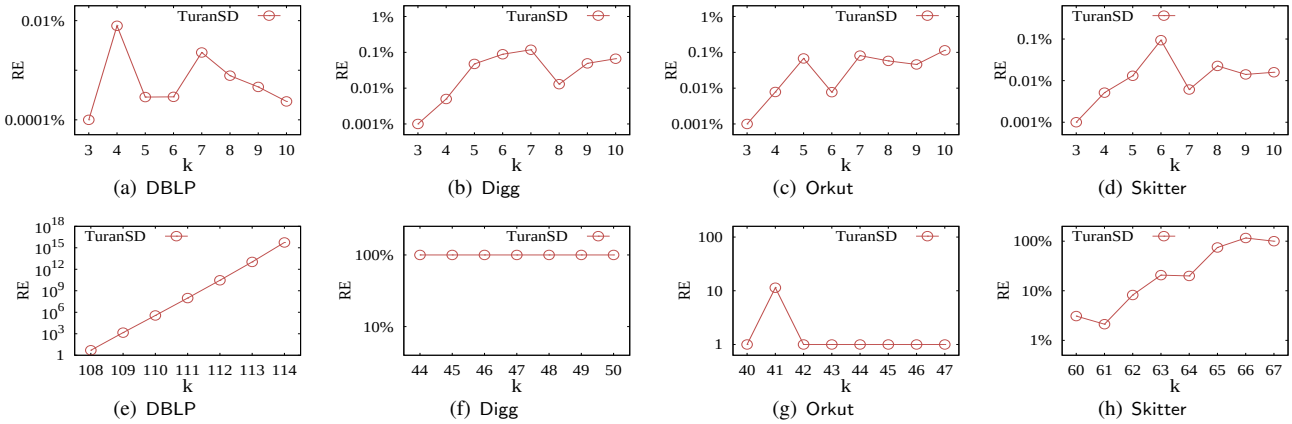


Fig. 12. Relative error (RE) of the Turán-Shadow algorithm on large-ω graphs (vary k)

is unstable. In particular, TuranSD performs very well on Nasasrb and SocFBb, and yet it performs very bad on WikiCN and BaiduBK. More interestingly, we observe that the relative error of TuranSD seems to increase as k increases on most datasets. Note that this observation was not explicitly revealed in the previous studies [11], [12]. The reason could be that the success probability of the sampling procedure in TuranSD (see line 19 of Algorithm 5) decreases when k increases, thus reducing the estimating precision of the TuranSD algorithm. Similar results can also be observed on the large-ω graphs. In particular, Fig. 12 plots the relative error of TuranSD as a function of k on large-ω graphs. As can be seen, for small k values (Figs. 12(a-d)), the relative error of TuranSD typically increases with an increasing k. For large k-values, TuranSD performs extremely bad on all datasets. For example, the relative error of TuranSD on DBLP is more than 10^{12} for $k = 113$ which is clearly meaningless. These results indicate that TuranSD is very accurate to estimate the k-clique count for small k values, but it might be unreliable for estimating the count of large k-cliques.

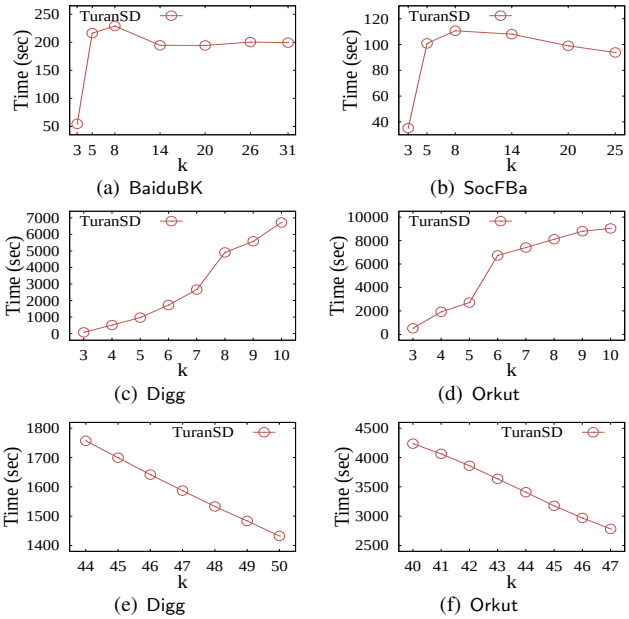


Fig. 13. Running time of the Turán-Shadow algorithm

Running time of TuranSD**.** Fig. 13 shows the running time of TuranSD with varying k. Note that previous studies in [11], [12] did not evaluate how the parameter k affects the running time of the TuranSD algorithm. As shown in Figs. 13(a-b), the running time of TuranSD seems to be robust w.r.t. k (except $k = 3$) on the small-ω graphs. However, on the large-ω graphs, its running time generally increases as k increases for small k (see Figs. 13(c-d)). This is because on large-ω graphs, the running time of TuranSD is dominated by the Turán-Shadow construction procedure. For a small k (e.g., $k \leq 10$), the size of the Turán-Shadow on large-ω graphs generally increases as k increases, thus the time for constructing the Turán-Shadow increases. In Figs. 13(e-f), we observe that if k is near to ω, the running time achieved by TuranSD decreases as k increases. The reason is that for a large k (near to ω), the Turán-Shadow size drops when k increases. Note that the running time of TuranSD is much faster than that of the optimized ordering-based algorithm (DDegCol) on large-ω graphs. For example, on Digg, TuranSD takes 5,583 seconds when $k = 9$, while DDegCol consumes 10,031 seconds. These results suggest that TuranSD can quickly obtain good estimations of k-clique counts for small k values on large-ω graphs. However, for a relatively large k (e.g., $k \geq 10$), TuranSD is also very costly

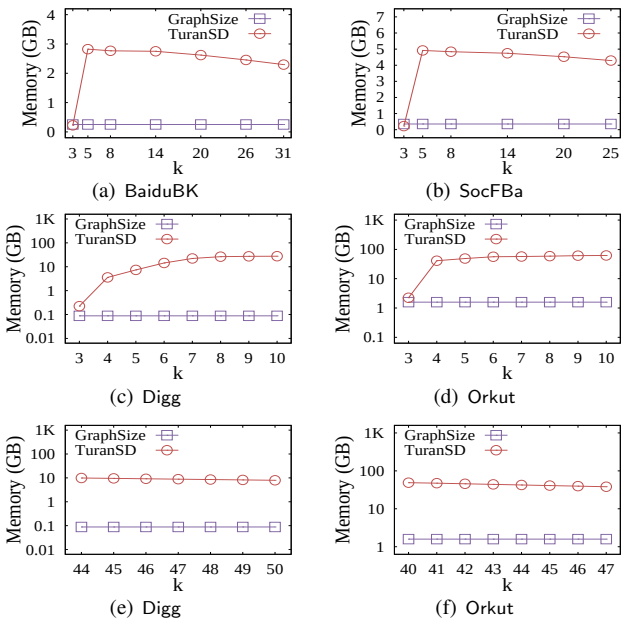


Fig. 14. The Turán-Shadow size with varying k

on large-ω graphs.

Size of the Turán-Shadow. Fig. 14 shows the size of the Turán-Shadow as a function of k. Note that the effect of the parameter k for the Turán-Shadow size was also not systematically studied in [11], [12]. As can be observed in Figs. 14(a-b), the Turán-Shadow size is robust w.r.t. k on the small-ω graphs (except $k = 3$), which are consistent with our results in Figs. 13(a-b). However, on the large-ω graphs, it increases as k increases from 3 to 10. For example, on Digg (Fig. 13(c)), the Turán-Shadow size is more than 100 times larger than the graph size when $k = 10$. As shown in Figs. 13(e-f), even when k is near to ω, the Turán-Shadow size is still very large which is around 100 times larger than the graph size on both Digg and Orkut. These results indicate that for a relatively large k, the space overhead of TuranSD is very high on large-ω graphs, which may prevent it to handle real-world large-ω graphs.

G. Summary of Experimental Results

As a summary, we have the following observations from our experimental results. First, all the ordering-based algorithms are significantly faster than the classic Chiba-Nishizeki algorithm. Second, the two existing ordering based algorithms achieve similar performance for listing k-cliques with a small value of k. Both of these two algorithms are inferior to the color-ordering based algorithms on the small-ω graphs. Third, on the large-ω graphs, the optimized color-ordering based algorithm is the most efficient algorithm compared to the other algorithms. Furthermore, it can also be used to list all k-cliques when k is near to a maximum clique size. The two existing ordering-based algorithms and the classic Chiba-Nishizeki algorithm, however, cannot be used in this case. Fourth, all the ordering-based algorithms with the EdgeParallel strategy can achieve a good degree of parallelism. Again, the parallel variant of the optimized color-ordering based algorithm is the winner compared to the other parallel algorithms. Finally, the Turán-Shadow algorithm is highly accurate for estimating the k-clique count with a small k and it is often significantly faster than all the exact algorithms, but this comes at the cost of considerable space overhead.

V. Related Work

k-clique listing and counting. Except the k-clique listing algorithms described in Section III, there exist several other algorithms that do not evaluate in this paper, since those algorithms are shown to be less efficient than the algorithms evaluated. Makino and Uno [29] proposed a maximal clique enumeration algorithm which can also be adapted to list k-cliques. Their algorithm, however, was shown to be less efficient than the algorithm proposed by Danisch et al. [12], thus it is precluded from being considered in this paper. More recently, in the theoretical community, Eden et al. [30] developed a sub-linear time approximation algorithm for k-clique counting. Their algorithm, however, is based on several complicated techniques which may be of theoretical interests. The k-clique listing problem was also studied in the MapReduce setting [28]. In particular, Finocchi et al. [28] proposed a MapReduce algorithm for k-clique listing based on a degree ordering heuristics, where the degree ordering heuristics has already been evaluated in this paper.

Our work is closely related to the triangle listing problem, as a triangle is 3-clique. In the literature, there exist a large number of algorithms for listing (or counting) triangles in a graph. Notable examples include [15], [16], [31]–[38]. Specifically, Schank [31] developed several efficient main-memory algorithm for triangle listing with time complexity $O(\alpha m)$. Latapy [32] proposed a compact-forward algorithm which has running time of $O(m\sqrt{m})$ in the worst case. Ortmann and Brandes [16] proposed an unified ordering based framework for triangle listing. Tsourakakis et al. [34] developed a sampling algorithm to estimate the number of triangles in a graph. Recently, several algorithms have been proposed to handle the case when the graph does not fit into the main memory. Becchetti et al. [33] devised a triangle counting algorithm in the semi-streaming model. Chu and Cheng [35], and Hu et al. [36] proposed I/O-efficient algorithms for triangle listing. Suri and Vassilvitskii [37], and Kolda et al. [38] developed triangle listing algorithms in the MapReduce setting.

Maximal clique enumeration. Our work is also related to the maximal clique enumeration problem. As an important graph mining operator, maximal clique enumeration technique has been applied in numerous real-world applications [19], [39]–[42]. Practical algorithms for maximal clique enumeration are the classic Bron-Kerbosch algorithm [39] and its variants [19], [40]. Tomita et al. [40] shown that the Bron-Kerbosch algorithm with a greedy pivoting strategy is essentially optimal for enumerating all maximal cliques. Eppstein et al. [19] developed an improved Bron-Kerbosch algorithm based on a degeneracy ordering heuristics. Cheng et al. devised an I/O-

efficient maximal clique enumeration algorithm for handling disk-resident graphs [41]. The same group also developed a parallel maximal clique enumeration algorithm using limited memory [42]. The maximal clique enumeration problem has also been investigated for some special types of graphs. For example, Viard et al. [43] studied a problem of listing all maximal cliques in temporal graphs, where each edge in the graph has a timestamp. Li et al. [44] proposed a maximal signed clique enumeration algorithm for signed graphs, where each edge in the graph can be positive or negative.

VI. Conclusion

Listing k-cliques in a graph is an important graph mining problem which has been widely used in community detection and social network analysis applications. In this paper, we present an experimental comparison of 8 various algorithms for k-clique listing/counting, using a variety of real-world graphs with up to 3 million nodes and 100 million edges. Based on our experimental results, we have the following observations. First, both the degree-based ordering algorithm and the degeneracy-ordering based algorithm work well for listing k-cliques with a small value of k, and yet they are inferior to the optimized color-ordering based algorithm. Second, all the three color-ordering based algorithms outperform the other algorithms on the small-ω graphs, and they also work well for listing k-cliques when k is near to a maximum clique size. Third, on the large-ω graphs, the optimized color-ordering based algorithm is the most efficient technique, suggesting that such an algorithm is a preferable choice in practical applications. Finally, the Turán-Shadow algorithm can quickly obtain a good approximation for the k-clique count with a small value of k. However, the space overhead of such an algorithm increases as k increases. As a consequence, the Turán-Shadow algorithm is more preferable only when the applications need an approximate k-clique count with a small k.

Several future directions on k-clique listing/counting that may deserve further investigation. First, the color ordering heuristics could also be applied to improve the Turán-Shadow algorithm, in which the DAG is constructed by a degeneracy ordering. With a DAG generated by the color ordering heuristics, we can also prune the unpromising search paths in the Turán-Shadow algorithm, which reduces the construction time of the Turán Shadow. Second, all the techniques discussed in this paper assume that the graph can be fitted into the main memory. An interesting question is how to devise an I/O-efficient k-clique listing algorithm when the graph cannot be stored in the main memory. Finally, it may be interesting to apply the proposed color-ordering based techniques to improve the performance of the algorithms for k-clique-percolation based community detection [7], [45], large near clique detection [9], and k-clique densest subgraph problem [13].

References

[1] A. Angel, N. Koudas, N. Sarkas, and D. Srivastava, "Dense subgraph maintenance under streaming edge weight updates for real-time story identification," *PVLDB*, vol. 5, no. 6, pp. 574–585, 2012.

[2] R.-H. Li, J. X. Yu, and R. Mao, "Efficient core maintenance in large dynamic graphs," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 10, pp. 2453–2465, 2014.

[3] A. E. Sariyüce, C. Seshadhri, A. Pinar, and Ü. V. Çatalyürek, "Finding the hierarchy of dense subgraphs using nucleus decompositions," in *WWW*, 2015.

[4] L. Qin, R. Li, L. Chang, and C. Zhang, "Locally densest subgraph discovery," in *KDD*, 2015.

[5] R.-H. Li, L. Qin, J. X. Yu, and R. Mao, "Influential community search in large networks," *PVLDB*, vol. 8, no. 5, pp. 509–520, 2015.

[6] R. Li, L. Qin, F. Ye, J. X. Yu, X. Xiao, N. Xiao, and Z. Zheng, "Skyline community search in multi-valued networks," in *SIGMOD*, 2018.

[7] G. Palla, I. Derenyi, I. Farkas, and T. Vicsek, "Uncovering the overlapping community structure of complex networks in nature and society," *Nature*, vol. 435, no. 7043, 2005.

[8] E. Gregori, L. Lenzini, and S. Mainardi, "Parallel k-clique community detection on large-scale networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 8, pp. 1651–1660, 2013.

[9] M. Mitzenmacher, J. Pachocki, R. Peng, C. E. Tsourakakis, and S. C. Xu, "Scalable large near-clique detection in large-scale networks via sampling," in *KDD*, 2015.

[10] A. R. Benson, D. F. Gleich, and J. Leskovec, "Higher-order organization of complex networks," *Science*, vol. 353, no. 6295, 2016.

[11] S. Jain and C. Seshadhri, "A fast and provable method for estimating clique counts using turán's theorem," in *WWW*, 2017.

[12] M. Danisch, O. D. Balalau, and M. Sozio, "Listing k-cliques in sparse real-world graphs," in *WWW*, 2018.

[13] C. E. Tsourakakis, "The k-clique densest subgraph problem," in *WWW*, 2015.

[14] A. Angel, N. Koudas, N. Sarkas, and D. Srivastava, "Dense subgraph maintenance under streaming edge weight updates for real-time story identification," *PVLDB*, vol. 5, no. 6, pp. 574–585, 2012.

[15] N. Chiba and T. Nishizeki, "Arboricity and subgraph listing algorithms," *SIAM J. Comput.*, vol. 14, no. 1, pp. 210–223, 1985.

[16] M. Ortmann and U. Brandes, "Triangle listing algorithms: Back from the diversion," in *ALENEX*, 2014.

[17] C. S. J. A. Nash-Williams, "Decomposition of finite graphs into forests," *Journal of the London Mathematical Society*, vol. 39, no. 1, pp. 12–12, 1964.

[18] M. C. Lin, F. J. Soulignac, and J. L. Szwarcfiter, "Arboricity, h-index, and dynamic algorithms," *Theor. Comput. Sci.*, vol. 426, pp. 75–90, 2012.

[19] D. Eppstein, M. Löffler, and D. Strash, "Listing all maximal cliques in large sparse real-world graphs," *ACM Journal of Experimental Algorithmics*, vol. 18, 2013.

[20] P. Turan, "On an extremal problem in graph theory," *Mat. Fiz. Lapok*, vol. 48, no. 137, pp. 436–452, 1941.

[21] W. Hasenplaugh, T. Kaler, T. B. Schardl, and C. E. Leiserson, "Ordering heuristics for parallel graph coloring," in *SPAA*, 2014.

[22] L. Yuan, L. Qin, X. Lin, L. Chang, and W. Zhang, "Effective and efficient dynamic graph coloring," *PVLDB*, vol. 11, no. 3, pp. 338–351, 2017.

[23] S. B. Seidman, "Network structure and minimum degree," *Social Networks*, vol. 5, no. 3, pp. 269–287, 1983.

[24] V. Batagelj and M. Zaversnik, "An O(m) algorithm for cores decomposition of networks," *CoRR*, vol. cs.DS/0310049, 2003.

[25] H. N. Gabow and H. H. Westermann, "Forests, frames, and games: Algorithms for matroid sums and applications," *Algorithmica*, vol. 7, no. 5&6, pp. 465–497, 1992.

[26] L. Barenboim and M. Elkin, "Sublogarithmic distributed MIS algorithm for sparse graphs using nash-williams decomposition," in *PODC*, 2008, pp. 25–34.

[27] D. Eppstein and E. S. Spiro, "The h-index of a graph and its application to dynamic subgraph statistics," *J. Graph Algorithms Appl.*, vol. 16, no. 2, pp. 543–567, 2012.

[28] I. Finocchi, M. Finocchi, and E. G. Fusco, "Clique counting in mapreduce: Algorithms and experiments," *ACM Journal of Experimental Algorithmics*, vol. 20, pp. 1.7:1–1.7:20, 2015.

[29] K. Makino and T. Uno, "New algorithms for enumerating all maximal cliques," in *9th Scandinavian Workshop on Algorithm Theory*, 2004.

[30] T. Eden, D. Ron, and C. Seshadhri, "On approximating the number of k-cliques in sublinear time," in *STOC*, 2018.

[31] T. Schank, "Algorithmic aspects of triangle-based network analysis," Ph.D. dissertation, Universitat Karlsruhe (TH), 2007.

[32] 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.
[33] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis, "Efficient semi-streaming algorithms for local triangle counting in massive graphs," in *KDD*, 2008.
[34] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos, "DOULION: counting triangles in massive graphs with a coin," in *KDD*, 2009.
[35] S. Chu and J. Cheng, "Triangle listing in massive networks and its applications," in *KDD*, 2011.
[36] X. Hu, Y. Tao, and C. Chung, "I/o-efficient algorithms on triangle listing and counting," *ACM Trans. Database Syst.*, vol. 39, no. 4, pp. 27:1–27:30, 2014.
[37] S. Suri and S. Vassilvitskii, "Counting triangles and the curse of the last reducer," in *WWW*, 2011.
[38] T. G. Kolda, A. Pinar, T. D. Plantenga, C. Seshadhri, and C. Task, "Counting triangles in massive graphs with mapreduce," *SIAM J. Scientific Computing*, vol. 36, no. 5, 2014.
[39] C. Bron and J. Kerbosch, "Finding all cliques of an undirected graph (algorithm 457)," *Commun. ACM*, vol. 16, no. 9, pp. 575–576, 1973.
[40] 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.
[41] J. Cheng, Y. Ke, A. W.-C. Fu, J. X. Yu, and L. Zhu, "Finding maximal cliques in massive networks," *ACM Trans. Database Syst.*, vol. 36, no. 4, pp. 21:1–21:34, 2011.
[42] J. Cheng, L. Zhu, Y. Ke, and S. Chu, "Fast algorithms for maximal clique enumeration with limited memory," in *KDD*, 2012.
[43] J. Viard, M. Latapy, and C. Magnien, "Computing maximal cliques in link streams," *Theor. Comput. Sci.*, vol. 609, pp. 245–252, 2016.
[44] R.-H. Li, Q. Dai, L. Qin, G. Wang, X. Xiao, J. X. Yu, and S. Qiao, "Efficient signed clique search in signed networks," in *ICDE*, 2018.
[45] L. Yuan, L. Qin, W. Zhang, L. Chang, and J. Yang, "Index-based densest clique percolation community search in networks," *IEEE Trans. Knowl. Data Eng.*, vol. 30, no. 5, pp. 922–935, 2018.