Listing k-cliques in Sparse Real-World Graphs

Maximilien Danisch, Oana Balalau, Mauro Sozio

Listing k-cliques in Sparse Real-World Graphs*

Maximilien Danisch†
Sorbonne Université, CNRS,
Laboratoire d'Informatique de Paris 6,
LIP6, F-75005 Paris, France
maximilien.danisch@lip6.fr

Oana Balalau‡
Max Planck Institute for Informatics,
Saarbrücken, Germany
obalalau@mpi-inf.mpg.de

Mauro Sozio
LTCI, Télécom ParisTech University
Paris, France
sozio@telecom-paristech.fr

ABSTRACT

Motivated by recent studies in the data mining community which require to efficiently list all k-cliques, we revisit the iconic algorithm of Chiba and Nishizeki and develop the most efficient parallel algorithm for such a problem. Our theoretical analysis provides the best asymptotic upper bound on the running time of our algorithm for the case when the input graph is sparse. Our experimental evaluation on large real-world graphs shows that our parallel algorithm is faster than state-of-the-art algorithms, while boasting an excellent degree of parallelism. In particular, we are able to list all k-cliques (for any k) in graphs containing up to tens of millions of edges as well as all 10-cliques in graphs containing billions of edges, within a few minutes and a few hours respectively. Finally, we show how our algorithm can be employed as an effective subroutine for finding the k-clique core decomposition and an approximate k-clique densest subgraphs in very large real-world graphs.

CCS CONCEPTS

• **Mathematics of computing → Graph algorithms**;

KEYWORDS

k-clique listing and counting, real-world graph algorithms

1 INTRODUCTION

Finding dense subgraphs is an important research area in graph mining [19, 37], with applications in community detection [20], spam-link farms in web graphs [27], real-time story identification [5, 39], motif detection in biological networks [26], epilepsy prediction [30], graph compression [13], distance query indexing [32], finding correlated genes [45], finance [21] and many others.

Cliques are the dense subgraphs par excellence. The concept of a clique has been originally introduced by sociologists to measure social cohesiveness before the advent of computers [52]. In our work, we study the problem of listing all k-cliques in a graph, which are subgraphs with k nodes, each pair of which being connected with an edge. Such a problem is a natural generalization of the problem of listing triangles, which has been intensively studied by the research community. In particular, state-of-the-art algorithms can list all triangles in real-world graphs containing several billions of edges [36, 40, 49] within a few hours. In contrast, listing all k-cliques is often deemed not feasible with most of the works focusing on approximately counting cliques [31, 42].

Recent works in the data mining and database community call for efficient algorithms for listing or counting all k-cliques in the input graph. In particular, in [51] the author develops an algorithm for finding subgraphs with maximum average number of k-cliques, with listing k-cliques being an important building block. In [46] an algorithm for organizing cliques into hierarchical structures is presented, which requires to list all k-cliques. In [5], algorithms for finding cliques and quasi-cliques (i.e. cliques where a few edges might be missing) with at most k nodes are used for story-identification in social media. Algorithms for listing k-cliques (and more generally k-motifs) may also be used to reveal the latent higher-order organization in real-world graphs [8] or to compute percolated k-clique communities [28, 43].

Motivated by the aforementioned studies, we develop the most efficient algorithm for listing and counting all k-cliques in large sparse real-world graphs, with k being an input parameter. In fact, real-world graphs are often "sparse" and rarely contain very large cliques which allows us to solve such a problem efficiently. In our work, the sparsity of a graph is measured by its core value. We revisit the iconic algorithm developed by Chiba and Nishizeki [16], which is one of the most efficient algorithms for this problem. By means of several improvements on such an algorithm, we are able to provide the best asymptotic upper bound on the running time, in the case when the input graph has "low" core value. Moreover, our algorithm can be efficiently parallelized resulting in even better performances in practice.

Our extensive experimental evaluation shows that both the sequential and parallel versions of our algorithm outperform significantly state-of-the-art approaches for the same problem. In particular, our parallel algorithm is able to list all cliques in graphs containing up to tens millions edges, as well as all 10-cliques in graphs containing billions of edges, within a few minutes or a few hours, respectively, while achieving an excellent degree of parallelism. We show that our algorithm can be employed as an effective subroutine for computing a k-clique core decomposition in large graphs, an approximation of the k-clique densest subgraph [51], as well as, for finding quasi-cliques. Finally, we are able to estimate the accuracy of the approach proposed in [31] in approximating the number of k-cliques, for those graphs whose exact count of k-cliques was not known before our work.

The rest of the paper is organized as follows. We present the related work in Section 2 and the notations we use in Section 3. In Section 4, we present our algorithm for listing k-cliques and prove

*This research was partially supported by French National Agency (ANR) under project FIELDS (ANR-15-CE23-0006) and by a Google Faculty Award.
This work is also funded in part by the European Commission H2020 FETPROACT 2016-2017 program under grant 732942 (ODYCCEUS), by the ANR (French National Agency of Research) under grant ANR-15-CE38-0001 (AlgoDiv), by the Ile-de-France Region and its program FUI21 under grant 16010629 (iTRAC).
†Most of the work was done while the author was a postdoc at Télécom ParisTech.
‡Most of the work was done while the author was a student at Télécom ParisTech.

its theoretical guarantees in Section 5. We then evaluate the performance of our algorithm against the state-of-the-art (Section 6). In Section 6.5 we adapt our algorithm so as to compute the k-clique core decomposition of a graph and compute an approximation of the k-clique densest subgraph. In Section 6.6, we estimate the accuracy of the approach proposed in [31] for those graphs whose exact count of k-cliques was not known before our work. Finally, we conclude and present future work in Section 7.

2 RELATED WORK

We organize the related work into the following sections: listing all k-cliques, listing all maximal cliques, counting k-cliques, counting k-motifs and finding k-clique densest subgraphs. Given that the related work in this research area is vast, our related work might not be comprehensive and will focus only on the most relevant works to our study.

Listing all k-cliques. Prior to our work, the sequential algorithm with the best known asymptotic running time for listing k-cliques in sparse graphs was the algorithm of Chiba and Nishizeki [16], to the best of our knowledge. Its running time is in $O(k \cdot m \cdot a(G)^{k-2})$, where $a(G)$ is the arboricity of the graph. For our algorithm, we are able to provide an asymptotic upper bound of $O(k \cdot m \cdot (\frac{c(G)}{2})^{k-2})$, where $c(G)$ is the core value of the graph. Given that $c(G) \leq 2a(G) - 1$ for any G, our upper bound becomes better when k is large enough. We defer to future work, a more rigorous study of the running time of the two algorithms. In practice, our algorithm is nearly an order of magnitude faster, while our parallel algorithm achieves an optimal degree of parallelism. We provide an efficient implementation of [16] in C, which to the best of our knowledge was not publicly available.

The approach presented in [40, 49] has been proved to be efficient in practice. Although initially devised for counting and listing maximal cliques it can also be adapted to k-clique listing and counting. It is based on the well-known Bron-Kerbosch algorithm [12, 15, 40, 50] for counting maximal cliques. An efficient implementation of such an algorithm is available at [1]. According to our experiments, such an approach appears to be less efficient than our implementation of the algorithm of Chiba and Nishizeki.

Observe that those algorithms are sequential. Our experimental evaluation shows that one can highly benefit from uniformly distributing the computational load across several threads.

Triangle listing and counting (i.e. 3-clique listing) have been studied intensively in recent years, with the algorithms proposed in [16, 36, 40] being perhaps the most efficient ones in practice, when the input graph fits into main memory. In particular, the compact-forward algorithm developed in [36] has running time of $O(m \cdot \sqrt{m})$, while in the case when the degree distribution of the input graph G follows a power law distribution with exponent $\alpha(G)$ the running time is in $O(m \cdot n^{\frac{1}{\alpha(G)}})$. The algorithm presented in [16] has running time $O(m \cdot a(G))$, where $a(G)$ is the arboricity of the graph and m is the number of edges. In [40, 49], authors develop an algorithm to list all maximal cliques, while an implementation of their algorithm to count triangles efficiently is available at [1]. From a theoretical point of view, the state of the art algorithm for triangle counting and listing are [4] and [10], respectively. The running time of [4] is $O(m^{\frac{2\omega}{\omega+1}})$, where ω denotes the fast matrix multiplication exponent. The algorithms proposed in [10] are output sensitive, i.e., they are fast if the number of triangles in the input graph is small. It is unclear whether the two latter algorithms work well in practice. Several algorithms have been proposed in the case when the graph does not fit into main memory, such as [34] in the MapReduce architecture, [7] in the semi-streaming model as well as the I/O-efficient algorithm presented in [17]. Generally speaking, MapReduce algorithms seem to be slower than main-memory based algorithms, however, they could scale to larger graphs [42].

Listing all maximal cliques. Most algorithms for listing all maximal cliques are based on the seminal algorithm developed by Bron and Kerbosch [12, 15, 40, 50]. The state of the art algorithm for this problem has running time $O(c(G) \cdot n \cdot 3^{\frac{c(G)}{3}})$ [23], where $c(G)$ is the core number of the input graph. Such running time is almost tight as the largest number of maximal cliques in a graph with core number $c(G)$ is $(n - c(G)) \cdot 3^{\frac{c(G)}{3}}$. There are also efficient parallel versions of such an algorithm [47]. In practice, algorithms for maximal cliques enumeration or counting can deal with large real-world graphs but hardly scale to very large graphs [24]. Recent works have focused on devising distributed algorithms that can deal with large real-world graphs via a distributed computation on smaller blocks of data [15, 18].

Counting k-cliques. Listing all k-cliques in a graph is often deemed not feasible with most of the works focusing on approximately counting cliques [31, 42]. Armed with our efficient algorithm for listing all k-cliques in a graph, we are able to estimate the accuracy of the approach proposed in [31] for those graphs whose exact count of k-cliques was not known before our work. An algorithm for computing the exact count of k-cliques has been developed in [25] for the MapReduce framework.

Counting k-motifs. k-cliques are a special case of k-motifs, therefore we also include the related work for counting k-motifs (also called graphlets). In [2], the authors develop an algorithm for counting each of all possible 4-motifs in large graphs which was later generalized to 5-motifs [44]. In [11], the authors provide algorithms with statistical guarantees for the approximate counting of k-motifs. The former work is based on color coding [3], a randomized algorithm for finding simple paths or cycles of length k.

k-clique densest subgraph and core decomposition. In [51], the author studies the k-clique densest subgraph problem, which consists of finding a subgraph with maximum ratio between the number of its k-cliques and its number of nodes. This is a generalization of the well-known densest subgraph problem, which has received increasing attention in recent years [6, 19, 22]. In [51], the author develops an algorithm to compute the k-clique core decomposition, which is a generalization of the well-known core decomposition. In our work, we show how to significantly speed up the computation of the two aforementioned problems by means of our algorithm for listing k-cliques.

3 DEFINITIONS AND NOTATION

We assume w.l.g. that the input graph is connected with its size being $O(m)$, where m is the number of edges. Otherwise, our algorithm can be executed on each of the connected components, separately. We denote with $V(G)$ and $E(G)$ the set of nodes and the set of edges of an undirected graph G, respectively. We denote with $\Delta_G(u)$ the set of neighbors of a node u in G, while $G[S]$ denotes the subgraph induced by the set of nodes $S \subseteq V(G)$.

We shall use the notation $\vec{G}$ to denote a directed graph (where the set of edges are ordered pairs). A Directed Acyclic Graphs (DAG) is a directed graph with no directed cycles. We denote with $\Delta^+_{\vec{G}}(u)$ and $\Delta^-_{\vec{G}}(u)$ the set of out-neighbors and the set of in-neighbors of node u in $\vec{G}$, respectively, while $\delta^+(u)$ and $\delta^-(u)$ denote the number of out-neighbors and in-neighbors of u, respectively. We shall only refer to $\Delta^+_{\vec{G}}(u)$ in the rest of our paper, therefore we shall use the notation $\Delta_{\vec{G}}(u) := \Delta^+_{\vec{G}}(u)$. Similarly to the case of undirected graphs, $\vec{G}[S]$ denotes the induced subgraph by the set of nodes S in $\vec{G}$.

The *arboricity* $a(G)$ of a graph G is defined as the minimum number of forests into which the set of edges of G can be partitioned. The *core value* $c(G)$ (also called *degeneracy*) of a graph G is defined as the maximum integer c such that there exists an induced subgraph H of G with all nodes having degree at least c. It is known that for any graph G, $c(G)$ is in $[a(G), 2a(G) - 1]$ ([53]).

The core value of a graph can be computed in linear time by the algorithm that repeatedly removes a node with minimum degree until the graph becomes empty. We shall refer to any ordering on the nodes induced by such an algorithm, as *core ordering*.

Let G be an undirected graph, let $\eta : V(G) \mapsto [1, |V(G)|]$ be a total ordering on the nodes of G and let $\vec{G}$ be a directed graph. We say that $\vec{G}$ is induced by the ordering η, if $V(G) := V(\vec{G})$ and there is an edge $v \to u$ if $\eta(v) < \eta(u)$ and $(u, v) \in E(G)$. Observe that $\vec{G}$ is a DAG. Moreover, observe that, if η is the core ordering, then the maximum out-degree in $\vec{G}$ is the core value of G.

Our theoretical analysis is in the worst case, unless otherwise specified. We study the *parameterized complexity* of our problem. We measure the running time of our algorithm as a function of the input size m, as well as the parameters k and $c(G)$ which significantly affect its running time.

4 ALGORITHMS

4.1 Algorithm of Chiba and Nishizeki

We start by providing a compact description of the algorithm developed by Chiba and Nishizeki for listing k-cliques [16]. The algorithm processes the nodes in non-increasing order of degree. For each node u, the algorithm computes the subgraph induced by its neighbors, and then it recurses on such a subgraph. When processing the nodes of a given subgraph, its nodes might have to be reordered in non-increasing order of degree (line 7 of Algorithm 1). After processing a node u, u is deleted from the current graph so as to prevent that any clique containing u is listed more than once (line 10 of Algorithm 1). A pseudocode of the algorithm is shown in Algorithm 1.

Such an algorithm is relatively simple, yet it turns out to be one of the most efficient algorithms for listing k-cliques in sparse real-world graphs. This is obtained by using some efficient data structures, in particular, to compute the subgraphs induced on the neighbors of a given node (line 9 of Algorithm 1). An appealing feature of the algorithm is that its running time can be bounded as a function of the arboricity of the input graph, which is a natural way to measure the sparsity of a graph.

Theorem 4.1. *(Running time of Algorithm 1 ([16])) Algorithm 1 lists all k-cliques in $O(k \cdot m \cdot a(G)^{k-2})$ time (where $a(G)$ is the arboricity of the input graph G), while it requires linear memory.*

Algorithm 1 Algorithm of Chiba and Nishizeki [16]

```
1:  LISTING(k,G,∅)
2:  function LISTING(l,G,C)
3:      if l = 2 then
4:          for each edge (u,v) of G do
5:              output k-clique C ∪ {u,v}
6:      else
7:          u_1,...,u_|V(G)| ← nodes in G s.t. δ(u_i) ≥ δ(u_{i+1})
8:          for i = 1,...,|V(G)| do
9:              LISTING(l − 1,G[Δ_G(u_i)],C ∪ {u_i})
10:             V(G) ← V(G) \ {u_i}
```

We shall focus our efforts on improving the following three aspects of Algorithm 1.

(1) Can the algorithm be efficiently parallelized? Line 10 of Algorithm 1 introduces a sequential dependency between the nodes in the input graph, which makes an efficient parallelization of the algorithm non-straightforward. In particular, it does not seem trivial to balance the workload across the different threads in a multicore implementation.
(2) Is there a more efficient way of processing the nodes of the input graph? Algorithm 1 processes the nodes by non-increasing degree. Although this simplifies the analysis of the running time of the algorithm, it might not be an optimal ordering in practice.
(3) Can the number of operations be decreased even further? In particular, we are planning to consider a directed version of the input graph which shall guide us in avoiding unnecessary computations.

4.2 The kClist Algorithm

A pseudocode of our algorithm, called kClist, is shown in Algorithm 2. Observe the following differences. Our algorithm can receive in input any total ordering η on the nodes of the graph. According to our analysis and experiments, the core ordering is an ordering that performs well, however, other orderings on the nodes might be considered. A theoretical study on the best ordering is given in Section 5. Observe that such an ordering is used to obtain a DAG. The function LISTING is then modified as nodes are not sorted again inside the function (line 7 of Algorithm 1). Moreover, the DAG $\vec{G}$ prevents that a same clique is listed more than once, which makes the deletion step in line 10 of Algorithm 1 not necessary anymore.

This in turn, allows for a more efficient parallel implementation of our algorithm.

Algorithm 2 Our algorithm for listing k-cliques

1: Let η be a total ordering on the nodes of the input graph G
2: $\vec{G} \leftarrow$ directed version of G, where $v \rightarrow u$ if $\eta(v) < \eta(u)$
3: LISTING(k,$\vec{G}$,$\emptyset$)
4: **function** LISTING(l,$\vec{G}$,C)
5: **if** $l = 2$ **then**
6: **for** each edge (u,v) of $\vec{G}$ **do**
7: **output** k-clique $C \cup \{u,v\}$
8: **else**
9: **for** each node $u \in V(\vec{G})$ **do**
10: LISTING($l-1$,$\vec{G}[\Delta_{\vec{G}}(u)]$,$C \cup \{u\}$)

Efficient Implementation. The algorithm can be implemented efficiently using the following data structures and operations which are an adaptation of the ones used in the algorithm of Chiba and Nishizeki [16]. The DAG $\vec{G}$ created in line 2 is represented as an adjacency list storing for each node u, its out-degree $\delta^+(u)$ and all its out-neighbors in an array $\Delta(u)$. No other adjacency lists will be created. In particular, given the current graph $\vec{G}$ in the recursion, we make sure that the out-neighbors of any node v in $\vec{G}$ always appear first in the array $\Delta(v)$. This is obtained by reordering each time inside the recursion, which can be done efficiently. Given $\vec{G}$, $\vec{G}[\Delta_{\vec{G}}(u)]$ is built as follows.

- Use an array with a label for each node initially set to k.
- For each out-neighbor v of node u in $\vec{G}$, set its label to $l-1$ if the label was equal to l. We thus have that if a label of a node v is equal to $l-1$ it means that node v is in the new DAG $\vec{G}[\Delta_{\vec{G}}(u)]$.
- For each out-neighbor v of u, move all the out-neighbors of v with label equal to $l-1$ in the first part of $\Delta(v)$ (by swapping nodes) and compute the out-degree of node v in the new DAG $\vec{G}[\Delta_{\vec{G}}(u)]$ updating $d(v)$. The first $d(v)$ nodes in $\Delta(v)$ are thus the out-neighbors of v in $\vec{G}[\Delta_{\vec{G}}(u)]$.

4.3 Efficient Parallel Algorithm

One appealing feature of our algorithm is that the $\vec{G}[\Delta_{\vec{G}}(u)]$'s can be processed independently. The same clique would not be listed more than once, thanks to the fact that $\vec{G}$ is a directed graph. Such a problem has been circumvented in Algorithm 1 by line 10 (where each node is deleted after being processed), which unfortunately makes an efficient parallelization non-trivial. It does not seem straightforward to achieve good workload balancing across the threads.

We refer to the variant of our algorithm where the $\vec{G}[\Delta_{\vec{G}}(u)]$'s are processed in parallel as NODE-PARALLEL. It turns out that such a first attempt in parallelizing the algorithm suffers from a poor degree of parallelism, in particular when the number of threads is large (in our experiments larger than 10). This is due to the fact that the number of k-cliques in the input graph might not be distributed uniformly across the $\vec{G}[\Delta_{\vec{G}}(u)]$'s, resulting in an unbalanced workload across the threads.

Such a problem is alleviated in what we call the EDGE-PARALLEL variant of our algorithm. In such a variant, each subgraph $\vec{G}[\Delta_{\vec{G}}(uv)]$ is processed independently in parallel, where $\Delta_{\vec{G}}(uv) = \Delta_{\vec{G}}(u) \cap \Delta_{\vec{G}}(v)$ denotes the set of out-neighbors of u and v with $(u,v) \in E(G)$. Since the $\vec{G}[\Delta_{\vec{G}}(uv)]$'s are "smaller" than the $\vec{G}[\Delta_{\vec{G}}(u)]$'s, we achieve a higher degree of parallelism. A pseudocode is shown in Algorithm 3, where the function LISTING has been defined in Algorithm 2. A pseudocode for NODE-PARALLEL can be easily obtained by replacing LISTING($k-2$,$\vec{G}[\Delta_{\vec{G}}(uv)]$,$\{u,v\}$) with LISTING($k-1$,$\vec{G}[\Delta_{\vec{G}}(u)]$,$\{u\}$) in line 4 of Algorithm 3 (and looping in parallel over the nodes instead of the edges in line 3). Note that in the edge-parallel version when $k = 3$ (i.e. we seek to list triangles), we need to modify the recursive function LISTING so that it outputs triangles and terminates when $l = 1$ and not $l = 2$ (line 5 of Algorithm 2).

At any point in time, each thread processes one of the $\vec{G}[\Delta_{\vec{G}}(uv)]$ subgraphs. Therefore, for each thread we need additionally $O(c(G)^2)$ space, in the case when the core ordering is used.

In Section 6 we evaluate both the NODE-PARALLEL and EDGE-PARALLEL variants of our algorithm, showing that EDGE-PARALLEL boasts an excellent degree of parallelism on 40 threads or more.

Algorithm 3 Our parallel algorithm (EDGEPARALLEL)

1: Let η be a total ordering on the nodes of the input graph G
2: $\vec{G} \leftarrow$ directed version of G, where $v \rightarrow u$ if $\eta(v) < \eta(u)$
3: **for** each edge $(u,v) \in E(\vec{G})$ in parallel **do**
4: LISTING($k-2$,$\vec{G}[\Delta_{\vec{G}}(uv)]$,$\{u,v\}$)

5 ANALYSIS OF THE ALGORITHM

LEMMA 5.1. *(Correctness) Algorithm 2 lists every k-clique exactly once.*

PROOF. Let $v_1, \ldots, v_k$ be the nodes of a k-clique. Suppose w.l.g that $\forall l \in [\![2,k]\!]$, $\eta(v_{l-1}) > \eta(v_l)$. Observe that this is the only ordering such that $\forall l \in [\![3,k]\!]$, $v_{l-1} \in \Delta_{\vec{G}}(v_l)$ (line 10) and $v_1 \in \Delta_{\vec{G}}(v_2)$ (line 7) . Therefore, such a k-clique is produced in output exactly once. □

LEMMA 5.2. *Let $\vec{G}$ be a DAG. Let C be any subset of the nodes in $\vec{G}$ with $|C| \leq (k-2)$. The running time of the function LISTING(l,$\vec{G}$,C) in Algorithm 2 can be upper bounded by $T(l,\vec{G})$ for which we can write the following recurrence:*

$$\begin{cases} T(2,\vec{G}) = \Theta(k \cdot |E(\vec{G})|) \\ T(l,\vec{G}) = \Theta\left(\sum_{u \in V(\vec{G})} \sum_{v \in \Delta_{\vec{G}}(u)} |\Delta_{\vec{G}}(v)|\right) + \sum_{u \in V(\vec{G})} T\left(l-1, \vec{G}[\Delta_{\vec{G}}(u)]\right). \end{cases}$$

PROOF. The time to output a k-clique for each edge in $\vec{G}$ (line 7) is $\Theta(k \cdot |E(\vec{G})|)$. The algorithm builds $\vec{G}[\Delta_{\vec{G}}(u)]$ for all u in $V(\vec{G})$ (line 10) which requires $\Theta(\sum_{u \in V(\vec{G})} \sum_{v \in \Delta_{\vec{G}}(u)} |\Delta_{\vec{G}}(v)|)$ time and then it recurses on each such a subgraph. □

Let G be an undirected graph. Let η be a total ordering on the nodes of G, while let $\vec{G}$ be the directed version of G induced by η.

From Lemma 5.2, it follows that the running time of Algorithm 2 with input G is $O(T(k,\vec{G}) + m)$.

It turns out that the total ordering η, used in our algorithm (line 1), might have a significant impact on its running time. Therefore, a natural goal is to determine an optimal such ordering. To this end, we give a simple upper bound on the running of our algorithm for the case of listing triangles. We show that the problem of finding an ordering which minimizes such an upper bound is NP-hard even for the case when $k = 3$.

Lemma 5.3. *(Running time for triangles) Let G be an undirected graph, let η be a total ordering on the nodes of G, while let $\vec{G}$ be the directed graph induced by such ordering. Algorithm 2 lists all triangles in $\Theta(m + \sum_{u \in V(\vec{G})} \delta^+_{\vec{G}}(u) \cdot \delta^-_{\vec{G}}(u))$.*

Proof. From Lemma 5.2, $k = 3$, it follows that the running time of Algorithm 2 is $\Theta(T(3, G_3) + m)$ where $T(3, G_3)$ is in $\Theta(\sum_{u \in V(\vec{G})} \sum_{v \in \Delta_{\vec{G}}(u)} |\Delta_{\vec{G}}(v)|)$. The claim follows from the fact that each term $|\Delta_{\vec{G}}(v)|$ occurs exactly $\delta^-_{\vec{G}}(u)$ times in the latter sum. □

We can then state our NP-hardness theorem.

Theorem 5.4. *(Minimizing the running time is NP-hard) Given a graph G, the problem of finding a total ordering η of the nodes in G which minimizes $\sum_{u \in V(\vec{G})} \delta^+_{\vec{G}}(u) \cdot \delta^-_{\vec{G}}(u)$ is NP-hard.*

Proof. (Sketch) Assume G is a 3-regular graph. Given an optimal ordering, a solution to Max Cut can be obtained in polynomial time from the ordering and as Max Cut in 3-regular graphs is NP-hard [9] the claim follows. We refer to [29] for a full proof. □

Given that it is unlikely that there is an efficient algorithm for finding such an optimal ordering, we consider the core ordering which is widely used in graph mining [14, 41]. Such an ordering is appealing in that it can be computed in linear time, while it produces an induced DAG with the smallest maximum out-degree (equals to the core value of the input graph). Therefore, in view of Lemma 5.2 it seems a natural choice. Using such an ordering for our algorithm we can obtain a smaller asymptotic upper bound on its running time than the one provided in [16] (when k is sufficiently large). An interesting direction for future work is to provide a tight analysis of the running time of the two algorithms.

The following two lemmas shall be useful in our main theorem.

Lemma 5.5. *Let $\vec{G}$ be the directed graph induced on G by a core ordering. The following holds:*

$$\sum_{u \in V(\vec{G})} \sum_{v \in \Delta_{\vec{G}}(u)} |\Delta_{\vec{G}}(v)| \leq c(G) \cdot |E(\vec{G})|.$$

Proof.

$$\sum_{u \in V(\vec{G})} \sum_{v \in \Delta_{\vec{G}}(u)} |\Delta_{\vec{G}}(v)| \leq \sum_{u \in V(\vec{G})} \sum_{v \in \Delta_{\vec{G}}(u)} c(G) \leq c(G) \cdot |E(\vec{G})|.$$

□

Lemma 5.6. *Let $\vec{G}$ be the directed graph induced on G by a core ordering. The following holds:*

$$\sum_{u \in V(\vec{G})} |E(\vec{G}[\Delta_{\vec{G}}(u)])| \leq |E(\vec{G})| \cdot \frac{c(G)}{2}.$$

Proof. Let $\vec{H}_u$ be the DAG induced by the out-neighbors of u in $\vec{G}$, that is, $\vec{H}_u = \vec{G}[\Delta_{\vec{G}}(u)]$. We have that:

$$\sum_{v \in \Delta_{\vec{G}}(u)} |\Delta_{\vec{H}_u}(v)| \leq \frac{1}{2} \cdot |\Delta_{\vec{G}}(u)| \cdot (|\Delta_{\vec{G}}(u)| - 1) < |\Delta_{\vec{G}}(u)| \cdot \frac{c(G)}{2}.$$

Hence, we can derive the following inequalities:

$$\sum_{u \in V(\vec{G})} \sum_{v \in \Delta_{\vec{G}}(u)} |\Delta_{\vec{H}_u}(v)| \leq \sum_{u \in V(\vec{G})} |\Delta_{\vec{G}}(u)| \cdot \frac{c(G)}{2} \leq |E(\vec{G})| \cdot \frac{c(G)}{2}.$$

□

We now have all the ingredients to prove our main theorem. We measure the running time of our algorithm as a function of the parameters k, $c(G)$ which significantly affect its running time.

Theorem 5.7. *Let G be a connected graph with m edges and core value $c(G)$. Algorithm 2 with core ordering lists all k-cliques in G in $O(k \cdot m \cdot (\frac{c(G)}{2})^{k-2} + m)$ time, while it requires linear space in the size of G.*

Proof. As we need to store only the input graph, $O(m)$ memory is sufficient. In the case when $c(G) = 1$ our algorithm terminates in linear time, in that, nodes in $\vec{G}$ have outdegree 1 or less. When $c(G) \geq 2$, our proof proceeds as follows. We prove by induction on l, $2 \leq l \leq k$, that $T(l, \vec{G}) \leq \lambda \cdot (k + \frac{l}{2}) \cdot (\frac{c(G)}{2})^{l-2} \cdot |E(\vec{G})|$, $\lambda > 0$.

For $l = 2$, it follows from Lemma 5.2 that the running time is at most $\lambda \cdot k \cdot |E(\vec{G})|$. For $l > 2$ we have:

$$\begin{aligned}
T(l,\vec{G}) &\leq \lambda \cdot c(G) \cdot |E(\vec{G})| + \sum_{u \in V(\vec{G})} T(l-1, \vec{G}[\Delta_{\vec{G}}(u)]) && (1)\\
&\leq \lambda \cdot c(G) \cdot |E(\vec{G})| \\
&+ \quad \lambda \cdot \sum_{u \in V(\vec{G})} (k + \frac{l-1}{2}) \cdot (\frac{c(G)}{2})^{l-3} \cdot |E(\vec{G}[\Delta_{\vec{G}}(u)])| && (2)\\
&\leq \lambda \cdot c(G) \cdot |E(\vec{G})| \\
&+ \quad \lambda \cdot (k + \frac{l-1}{2}) \cdot (\frac{c(G)}{2})^{l-2} \cdot |E(\vec{G})| && (3)\\
&\leq \lambda \cdot (k + \frac{l}{2}) \cdot (\frac{c(G)}{2})^{l-2} \cdot |E(\vec{G})|, && (4)
\end{aligned}$$

where Equation (2) follows from the inductive hypothesis, Equation (3) follows from Lemma 5.6, while Equation (4) follows from the fact that $c(G) \leq 2 \cdot (\frac{c(G)}{2})^{l-2}$ for $k, c(G) \geq 2, l > 2$.

□

As $a(G) \leq c(G) \leq 2 \cdot a(G) - 1$ our algorithm is in $O(k \cdot m \cdot (a(G) - \frac{1}{2})^{k-2})$ (for a connected graph) which is slightly better than the $O(k \cdot m \cdot a(G)^{k-2})$ bound provided by Chiba and Nishizeki. In graphs where $a(G) = c(G)$ the advantage is more significant.

We also derive an output sensitive bound on the running time that we express as a function of $c(G)$ and the number of l-cliques in the graph as shown in the following Theorem 5.8.

Lemma 5.8. *(Output sensitive bound) Algorithm 2 with core ordering lists all k-cliques in $O\Big(c(G) \cdot \sum_{l=2}^{k-1} N^l + k \cdot N^k\Big)$ time where N^l is the number of l-cliques.*

Proof. With Lemma 5.2 and Lemma 5.5, using core ordering, we have that for a given $l < k-2$ the time spent in the current level of recursion is $O(|E(\vec{G})| \cdot c(G))$ (the time to build all the $\vec{G}[\Delta_{\vec{G}}(u)]$), while for $l = k-2$ the time is in $O(k \cdot |E(\vec{G})|)$ (the time to output the $|E(\vec{G})|$ k-cliques). In addition, we notice that, for a given l, the sum of all $|E(\vec{G})|$ is exactly the number of $(k-l+2)$-cliques in the input graph G (the $k-l+2$ nodes in the set C), that is N^{k-l+2}. Summing it all we obtain the stated running time. □

Observe that to output all k-cliques, $\Omega(k \cdot N^k)$ operations are needed. The running time of our algorithm is thus optimal in the case when the term $k \cdot N^k$ dominates the term $c(G) \cdot \sum_{l=2}^{k-1} N^l$.

6 EXPERIMENTAL EVALUATION

6.1 Experimental Setup

We consider several real-world graphs that we obtained from [38]. We divide them into two main groups: *large* graphs containing up to tens of millions of edges, for which we are able to list all k-cliques, as well as *very large* graphs containing up to billions of edges and being less sparse for which we could only list k-cliques of limited size. Tables 1 and 2 summarize the statistics of the two groups of datasets, respectively.

For each large graph, we report its core number (which according to our theoretical analysis affects the running time of our algorithm), the size of a maximum clique and the number of maximum cliques. For each very large graph, we report its core number and the largest k for which we could list all k-cliques within one day of computation, as well as, the number of such k-cliques.

In our experiments we consider the core ordering, as it turns out to be more efficient in practice as well as from a theoretical point of view.

We carried our experiments on a Linux machine equipped with 4 processors Intel Xeon CPU E5- 2660 @ 2.60 GHz with 10 cores (a total of 40 threads) and with 64 G of RAM DDR4 2133 MHz. We evaluate both the sequential version of our algorithm, denoted as kClist1, and the parallel version of our algorithm denoted as kClist$n$, where n denotes the number of threads. We evaluate our method using 1, 10, and 40 threads (kClist1, kClist10 and kClist40, respectively) against the state-of-the-art for the exact listing of k-cliques[1]. In particular, we consider the following approaches:

- **CF:** the compact-forward algorithm for listing triangles of [36], for which we used the C implementation available at the webpage of the authors.
- **MACE:** the algorithm presented in [40, 49], for which we used the C implementation available at the webpage of the authors.
- **Arbo:** the algorithm of Chiba and Nishizeki presented in [16]. As we are not aware of any efficient implementation for such an algorithm, we provide our own implementation in C and made it publicly available.

[1]Our code is available at https://github.com/maxdan94/kClist.

6.2 Evaluation of our Sequential Algorithm

We start by comparing our sequential algorithm for listing triangles against the state-of-the-art algorithms for the same problem. Table 3 shows the running time of the algorithms for listing triangles (kClist1). We can see that even the sequential version of our algorithm outperforms the state-of-the-art algorithms for listing triangles. It also shows that the algorithm presented in [36] (CF) is very efficient for the task of listing triangles.

We then consider the k-clique listing problem on our collection of large datasets. Table 4 shows that our algorithm can list all k-cliques for any value of k in all those graphs, within a few minutes in most of the cases. It is faster than the other algorithms, up to a factor of 5. The full potential of our algorithm (including the parallel version) becomes apparent when listing k-cliques for larger k on very large graphs.

6.3 Degree of Parallelism

Our next step is to evaluate the degree of parallelism of our parallel algorithms. Ideally, we wish that when using t threads the running time decreases by a factor of t. This is measured by the *speedup*, which is defined as the running time of the sequential algorithm (with one thread) divided by the running time of the parallel algorithm when using t threads. We evaluate both the Node-Parallel and the Edge-Parallel variants of our algorithms, which are discussed in Section 4.3.

Figure 1 (left) shows the overall running time in linlog scale for Node-Parallel and Edge-Parallel for the problem of listing 8-cliques in DBLP, while Figure 1 (middle) shows the corresponding speedup. The running time required to perform I/O operations is also taken into account in that figure. We can see that Edge-Parallel boasts almost a linear speedup up to 40 threads, demonstrating that the computational load is always well balanced across the threads. In contrast, the speedup for Node-Parallel starts to worsen when the number of threads is larger than 10, with little benefit when using more than 20 threads.

This is consistent with what we observe in Figure 1 (right). Such a figure shows the percentage of CPU usage when using 40 threads, as a function of time. A percentage of 4000% means that all 40 threads are fully working. We can see that both Node-Parallel and Edge-Parallel manage to keep busy all 40 threads for the first 50 seconds. However, later on the percentage of CPU usage for Node-Parallel drops significantly, with very few threads being busy after two minutes. This is a clear sign that in this case the computational load is not well balanced across the threads.

We observe a similar behavior on the other datasets and for different values of k. We conclude that when few threads are available (say 10 or less), then a parallelization on the nodes is enough and leads to a nearly optimal speedup. When many threads are available

networks	n	m	c	k_{max}	$N_{k_{max}}$
road-CA	1,965,206	2,766,607	3	4	42
Amazon	334,863	925,872	7	7	32
soc-pocket	1,632,803	22,301,964	47	29	6
loc-gowalla	196,591	950,327	51	29	2
Youtube	1,134,890	2,987,624	51	17	2
cit-patents	3,774,768	16,518,947	64	11	2
zhishi-baidu	2,140,198	17,014,946	78	31	4
WikiTalk	2,394,385	4,659,565	131	26	141

Table 1: Our set of large graphs (for which we are able to list all k-cliques). c is the core value, k_{max} is the size of a maximum clique and $N_{k_{max}}$ is the number of maximum cliques.

networks	n	m	c	k	N_k
as-skitter	1,696,415	11,095,298	111	12	2.68×10^{14}
DBLP	425,957	1,049,866	113	11	8.23×10^{14}
Wikipedia	2,080,370	42,336,692	208	15	5.02×10^{14}
Orkut	3,072,627	117,185,083	253	12	4.15×10^{14}
Friendster	124,836,180	1,806,067,135	304	10	4.87×10^{14}
LiveJournal	4,036,538	34,681,189	360	7	4.49×10^{14}

Table 2: Our set of very large graphs (for which we are able to list k-cliques of limited size). k is the value till we could list all k-cliques within one day of computations using kClist40, N_k is the number of such k-cliques.

		Algorithms			
networks	# triangles	CF	MACE	Arbo	kClist1
as-skitter	28,769,868	5s	54s	5s	5s
DBLP	2,224,385	1s	4s	1s	1s
Wikipedia	145,707,846	37s	12m6s	3m10s	**29s**
Orkut	627,584,181	2m52s	17m15s	2m50s	**2m11s**
Friendster	4,173,724,142	1h50m41s	2h30m12s	2h48m16s	**1h0m7s**
LiveJournal	177,820,130	37s	3m23s	37s	**33s**

Table 3: Time to list triangles on our very large graphs.

	Algorithms		
networks	MACE	Arbo	kClist1
road-CA	1s	1s	1s
Amazon	1s	1s	1s
soc-pocket	18m15s	3m29s	**1m5s**
loc-gowalla	4m49s	1m38s	**31s**
Youtube	1m12s	6m7s	**3.4s**
cit-patents	16s	15s	**13s**
zhishi-baidu	33m17s	7m54s	**2m28s**
WikiTalk	>24h	7h42m27s	**1h45m33s**

Table 4: Time to list all cliques on our large graphs.

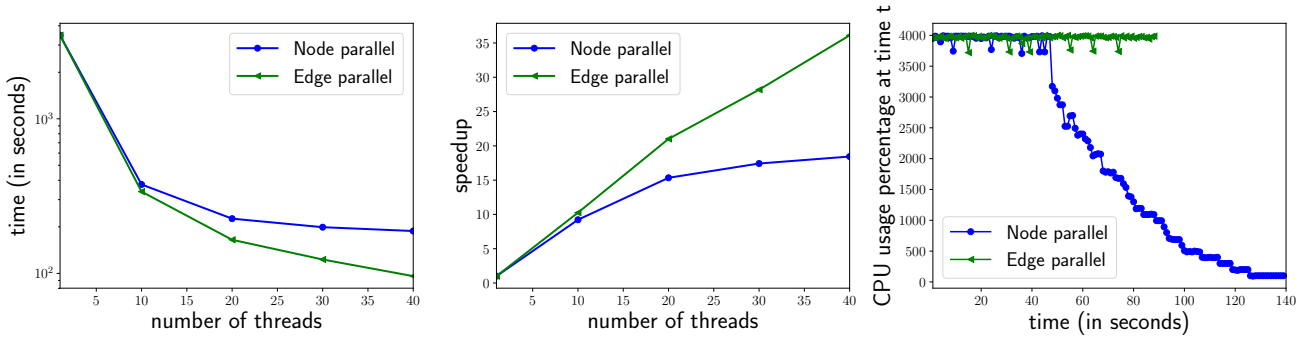

Figure 1: (left) Running time and (middle) speedup of our algorithms as a function of the number of used threads for listing 8-cliques in DBLP. (right) CPU percentage usage as a function of time using 40 threads for listing 8-cliques in DBLP.

then a parallelization on the edges is preferable. In what follows we use a parallelization on the edges for both kClist10 and kClist40.

6.4 k-clique Listing: Comparison

Figure 2 shows the running time of the algorithms as a function of k, when executed on the very large graphs. It shows that our sequential algorithm is significantly more efficient than the other approaches, with its running growing more gracefully as a function of k. For the largest values of k considered, it is at least one order of magnitude faster than [16], which is the most efficient algorithm after ours according to our experimental evaluation.

We can also see that the parallelization of our algorithm allows to deal with larger values of k (up to an additional term of 3), while requiring less than one day of computation. In particular, our parallel algorithm using 10 (resp. 40 threads) is the only one that can list all 9-cliques (resp. 10-cliques) in Friendster within a reasonable amount of time.

We also carried experiments in a Twitter graph [35], this graph has more than 1.6 billion edges and a high core value of 2647. Our parallel algorithm with 40 threads can list all 4-cliques within two hours of computations, while other approaches can only list triangles within a reasonable amount of time. It can also list all 3.39×10^{15} 5-cliques within a week of computation.

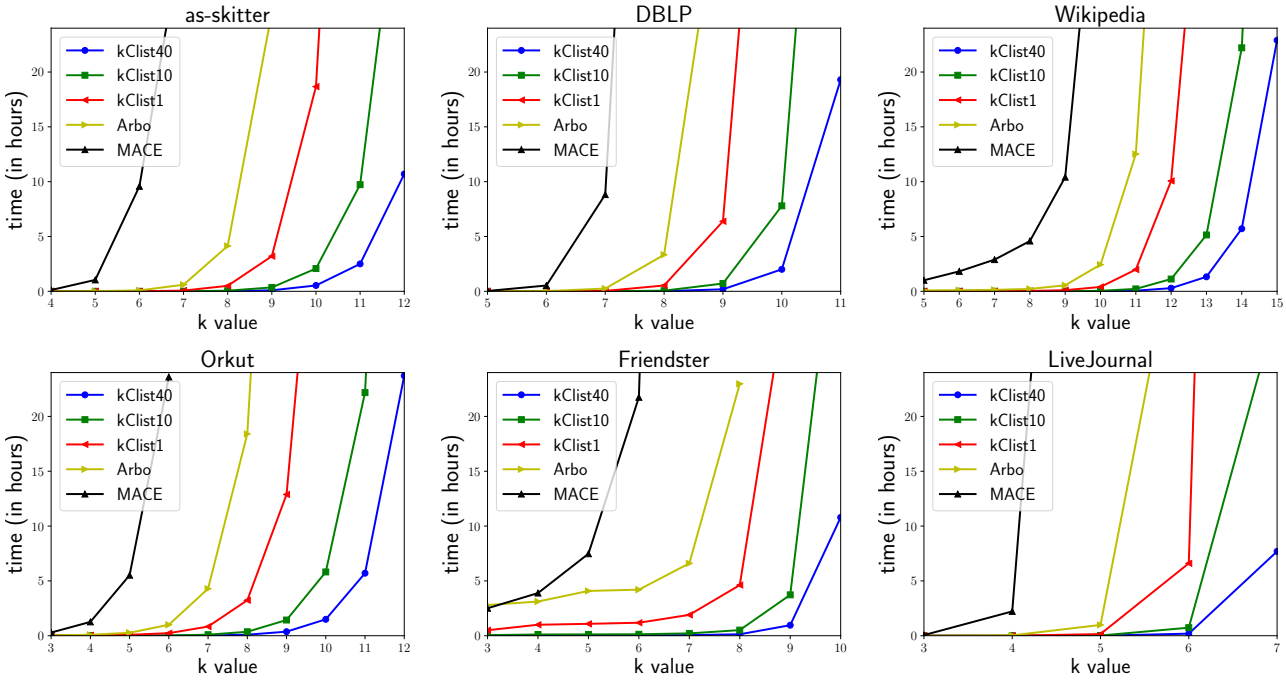

Figure 2: Running time of the algorithms as a function of k. We truncated the curves at 24 hours.

6.5 k-clique Densest Subgraph & k-clique Core Decomposition

We show that our algorithm can be used as an efficient subroutine to compute the *k-clique core decomposition* and an approximation to the k-clique densest subgraph, as defined in [51]. It turns out that using our algorithm one could solve those problems much faster, for larger values of k and on larger datasets than before. This, in turn, allows to efficiently find quasi-cliques, as pointed out in [51].

The k-clique density of a graph is defined as the number of k-cliques divided by the number of nodes in such a graph. The k-clique densest subgraph problem consists of finding a subgraph of the input graph with maximum k-clique density. As observed in [51], one could find an approximation to a k-clique densest subgraph by means of either one of the following two algorithms.

(1) The algorithm which removes in each round a vertex belonging to the minimum number of k-cliques and returns the subgraph that achieves the largest k-clique density, is a k-approximation for the k-clique densest subgraph problem. It also produces the so-called *k-clique core decomposition*.
(2) The algorithm which removes in each round the set of vertices belonging to less than $k(1+\epsilon)\rho_k$ k-cliques and returns the subgraph that achieves the largest k-clique density, where ρ_k is the k-clique density of the subgraph at that round. Such an algorithm gives a $k(1+\epsilon)$-approximation, while it terminates in $O(\frac{\log n}{\epsilon})$ round, for every $\epsilon > 0$.

Our algorithm can be used to give a full pass on all k-cliques of the input graph, without storing any such a clique. Therefore, it can be effectively employed to solve each of the aforementioned problems. In particular, a $k(1+\epsilon)$-approximation for the k-clique densest subgraph problem can be computed in $O(\log_{1+\epsilon}(n) \cdot k \cdot m \cdot (\frac{c(G)}{2})^{k-2} + m)$ time, while the k-clique core decomposition can be computed in $O(n \cdot k \cdot m \cdot (\frac{c(G)}{2})^{k-3} + m)$ time (in this version we update the k-clique degrees of the neighbors of a removed node by listing the $(k-1)$-cliques on the subgraph induced by its neighbors). Both algorithms require linear space in the size of the input.

In our experimental evaluation, using 40 threads, $\epsilon = 0.01$ we were able to compute a 13.13-approximation of the 13-clique densest subgraph in Wikipedia and a 9.09-approximation of the 9-clique densest subgraph in Friendster within one day of computation. In this case, we consider the $k(1+\epsilon)$-approximation algorithm for the k-clique densest subgraph which requires $O(\frac{\log n}{\epsilon})$ passes over the set of k-cliques. We were also able to compute within one day of computation while using one single thread, the 11-clique core decomposition of Wikipedia and the 7-clique core decomposition of Friendster.

The core decomposition ($k = 2$) is an effective tool in graph mining, for instance, to find best spreaders [33], find outliers [48], as well as speeding up algorithms (e.g. it is used as a subroutine in our own algorithm). An efficient algorithm for the k-clique core decomposition for $k > 2$ might unleash its full potential as a graph mining tool. In Figure 3, we show that such a decomposition can be used to efficiently find quasi-cliques. In such a figure, we measure the edge density of the k-approximation of the k-clique densest subgraph that we obtained using the k-clique core decomposition as a function of k, where the edge density of a subgraph is defined as its number of edges divided by the number of edges in a clique with the same number of nodes. We also report the number of nodes in the computed subgraph. We can see that the edge density of the

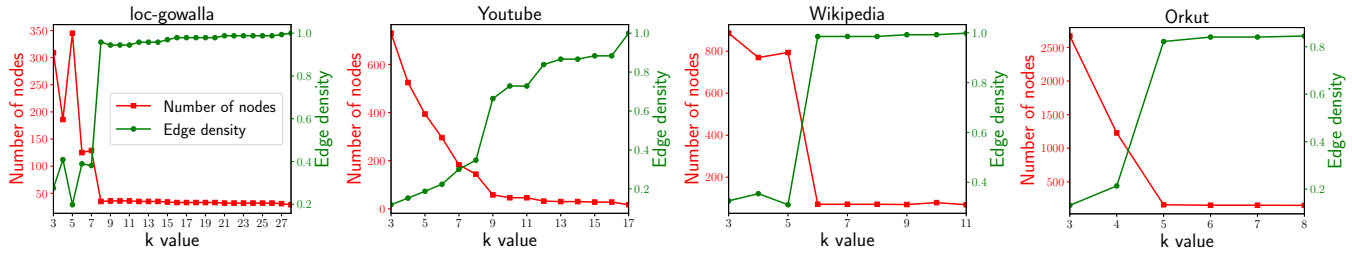

Figure 3: Number of nodes and edge density of the k-densest subgraph approximation versus k.

subgraphs found by our algorithm quickly converges towards 1, which is the density of a clique. This makes it an effective heuristic to find cliques and quasi-cliques in large graphs.

6.6 Counting k-cliques

Because of the computational bottlenecks in computing the exact number of k-cliques in a graph, there have been several methods for approximating such a number. Most notably, in [31] the authors developed a technique based on Turan's theorem, which is able to efficiently compute a good approximation of such a number. In particular, the authors show that their technique boasts an error of at most 2% in almost all the graphs considered in their work. However, for a few of those graphs, the exact count was not known at the time when their work was published.

Armed with our algorithm for listing all k-cliques in a graph, we are able to compute the exact number of k-cliques in all but two of the graphs considered in [31] and fill Table 2 of [31] with five additional values that were missing. The results are shown in Table 5. Our experiments confirm a very good accuracy of their technique, with all results boasting an error smaller than 1%[2] in terms of relative error (|true - estimate|/$true$). Moreover, the running time of their algorithm for approximately counting the number of k-cliques is faster than our algorithm for listing all k-cliques. In particular, all our experiments required at most 8 hours of computation with 40 threads, while the running time of their approach is in the order of minutes using a single thread, except for com-orkut which required three hours of computation. Our conclusion is that the technique proposed in [31] is valuable in case a quick approximate count is needed, however, our algorithm could be used in case one wishes to list all k-cliques or an exact count is needed.

All datasets and our implementation[3] are publicly available.

graph	m	k	N_k	Estimated N_k	error
web-Stanf.	1,992,636	10	5.8333×10^{12}	5.8358×10^{12}	0.04%
com-orkut	117,185,083	10	3.0288×10^{13}	3.0360×10^{13}	0.24%
com-lj	34,681,189	5	2.4663×10^{11}	2.4764×10^{11}	0.40%
		7	4.4902×10^{14}	4.5134×10^{14}	0.52%
as-skitter	11,095,298	10	1.4217×10^{13}	1.4312×10^{13}	0.67%

Table 5: Completion of Table 2 of [31]

[2]We remark that these results are significantly more accurate than initially reported due to a bug in their code, which has been later fixed by the authors.
[3]Available at https://github.com/maxdan94/kClist.

7 CONCLUSION AND FUTURE WORK

We developed a parallel algorithm for listing all k-cliques in very large real-world graphs which leverages the sparsity of the input graph. Our algorithm has the best known asymptotic running time, while it requires a linear amount of memory in the size of the input. In practice, for medium values of k, the sequential version of our algorithm is faster than state-of-the-art algorithms for the same problem, while the parallel version allows the gain of an order of magnitude with respect to state-of-the-art approaches. Our experimental analysis shows that our parallel algorithm is able to list all cliques in graphs containing up to tens of millions of edges, as well as all 10-cliques in graphs containing billions of edges, within a few minutes or a few hours, respectively, while boasting a near-optimal degree of parallelism. Observe that there are up to several quadrillions of k-cliques in the input graphs considered in our experiments. We showed that our algorithm can be used as an effective subroutine when computing the k-clique core decomposition or a k-clique densest subgraph. In particular, we showed how to use our algorithm so as to produce a stream of k-cliques without storing any such a clique in main memory. For future work, we would like to investigate further whether such an approach could be successfully employed in graph compression, community and event detection.

According to our theoretical analysis and experiments, the node ordering used in our algorithm significantly affects its running time, with core ordering performing well. One interesting direction for future work is to investigate whether one could speed up our algorithm even further by considering other node orderings. We showed that the edge-parallel version of our algorithm achieves a near-optimal degree of parallelism on up to 40 threads or more. A natural question is whether one could maintain or improve such a degree of parallelism on a larger number of threads. To this end, we speculate that one could benefit from parallelizing on higher order cliques (such as triangles or 4-cliques). We defer this study to future work.

Acknowledgements. We thank Qinna Wang for assistance with implementing a first version of the algorithm.

REFERENCES

[1] [n. d.]. http://research.nii.ac.jp/~uno/codes.htm. ([n. d.]).
[2] Nesreen K Ahmed, Jennifer Neville, Ryan A Rossi, and Nick Duffield. 2015. Efficient graphlet counting for large networks. In *Data Mining (ICDM), 2015 IEEE International Conference on*. IEEE, 1–10.
[3] Noga Alon, Raphael Yuster, and Uri Zwick. 1995. Color-coding. *J. ACM* 42, 4 (July 1995), 844–856. https://doi.org/10.1145/210332.210337

[4] Noga Alon, Raphael Yuster, and Uri Zwick. 1997. Finding and counting given length cycles. *Algorithmica* 17, 3 (1997), 209–223.

[5] Albert Angel, Nick Koudas, Nikos Sarkas, Divesh Srivastava, Michael Svendsen, and Srikanta Tirthapura. 2014. Dense subgraph maintenance under streaming edge weight updates for real-time story identification. *VLDB J.* 23, 2 (2014), 175–199.

[6] Oana Denisa Balalau, Francesco Bonchi, T.-H. Hubert Chan, Francesco Gullo, and Mauro Sozio. 2015. Finding Subgraphs with Maximum Total Density and Limited Overlap. In *WSDM*. 379–388.

[7] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis. 2008. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *SIGKDD*. 16–24.

[8] Austin R Benson, David F Gleich, and Jure Leskovec. 2016. Higher-order organization of complex networks. *Science* 353, 6295 (2016), 163–166.

[9] Piotr Berman and Marek Karpinski. 1999. On some tighter inapproximability results. In *International Colloquium on Automata, Languages, and Programming*. Springer, 200–209.

[10] Andreas Björklund, Rasmus Pagh, Virginia Vassilevska Williams, and Uri Zwick. 2014. Listing triangles. In *Automata, Languages, and Programming*. Springer, 223–234.

[11] Marco Bressan, Flavio Chierichetti, Ravi Kumar, Stefano Leucci, and Alessandro Panconesi. 2017. Counting Graphlets: Space vs Time. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining (WSDM '17)*. ACM, New York, NY, USA, 557–566. https://doi.org/10.1145/3018661.3018732

[12] Coen Bron and Joep Kerbosch. 1973. Algorithm 457: finding all cliques of an undirected graph. *Commun. ACM* 16, 9 (1973), 575–577.

[13] Gregory Buehrer and Kumar Chellapilla. 2008. A scalable pattern mining approach to web graph compression with communities. In *WSDM*. 95–106.

[14] Moses Charikar. 2000. Greedy approximation algorithms for finding dense components in a graph. In *Approximation Algorithms for Combinatorial Optimization*. Springer, 84–95.

[15] James Cheng, Linhong Zhu, Yiping Ke, and Shumo Chu. 2012. Fast algorithms for maximal clique enumeration with limited memory. In *SIGKDD*. ACM, 1240–1248.

[16] Norishige Chiba and Takao Nishizeki. 1985. Arboricity and subgraph listing algorithms. *SIAM J. Comput.* 14, 1 (1985), 210–223.

[17] Shumo Chu and James Cheng. 2011. Triangle listing in massive networks and its applications. In *SIGKDD*. 672–680.

[18] Alessio Conte, Roberto De Virgilio, Antonio Maccioni, Maurizio Patrignani, and Riccardo Torlone. 2016. Finding All Maximal Cliques in Very Large Social Networks Categories. (2016).

[19] Maximilien Danisch, T-H Hubert Chan, and Mauro Sozio. 2017. Large Scale Density-friendly Graph Decomposition via Convex Programming. In *Proceedings of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 233–242.

[20] Yon Dourisboure, Filippo Geraci, and Marco Pellegrini. 2009. Extraction and classification of dense implicit communities in the Web graph. *TWEB* 3, 2 (2009).

[21] Xiaoxi Du, Ruoming Jin, Liang Ding, Victor E. Lee, and John H. Thornton Jr. 2009. Migration motif: a spatial - temporal pattern mining approach for financial markets. In *SIGKDD*. 1135–1144.

[22] Alessandro Epasto, Silvio Lattanzi, and Mauro Sozio. 2015. Efficient Densest Subgraph Computation in Evolving Graphs. In *Proceedings of the 24th International Conference on World Wide Web, WWW 2015, Florence, Italy, May 18-22, 2015*. 300–310.

[23] David Eppstein, Maarten Löffler, and Darren Strash. 2010. *Listing all maximal cliques in sparse graphs in near-optimal time*. Springer.

[24] David Eppstein, Maarten Löffler, and Darren Strash. 2013. Listing all maximal cliques in large sparse real-world graphs. *Journal of Experimental Algorithmics (JEA)* 18 (2013), 3–1.

[25] Irene Finocchi, Marco Finocchi, and Emanuele G Fusco. 2015. Clique counting in mapreduce: Algorithms and experiments. *Journal of Experimental Algorithmics (JEA)* 20 (2015), 1–7.

[26] Eugene Fratkin, Brian T. Naughton, Douglas L. Brutlag, and Serafim Batzoglou. 2006. MotifCut: regulatory motifs finding with maximum density subgraphs. In *Proceedings 14th International Conference on Intelligent Systems for Molecular Biology 2006, Fortaleza, Brazil, August 6-10, 2006*. 156–157.

[27] David Gibson, Ravi Kumar, and Andrew Tomkins. 2005. Discovering Large Dense Subgraphs in Massive Graphs. In *PVLDB*. 721–732.

[28] Enrico Gregori, Luciano Lenzini, and Simone Mainardi. 2013. Parallel k-clique community detection on large-scale networks. *IEEE Transactions on Parallel and Distributed Systems* 24, 8 (2013), 1651–1660.

[29] Mikhail Rudoy (https://cstheory.stackexchange.com/users/34401/mikhail rudoy). [n. d.]. Is this vertex ordering optimization NP-Hard? Theoretical Computer Science Stack Exchange. ([n. d.]).

[30] Leonidas D. Iasemidis, Deng-Shan Shiau, Wanpracha Art Chaovalitwongse, J. Chris Sackellares, Panos M. Pardalos, Jose C. Principe, Paul R. Carney, Awadhesh Prasad, Balaji Veeramani, and Kostas Tsakalis. 2003. Adaptive epileptic seizure prediction system. *IEEE Trans. Biomed. Engineering* 50, 5 (2003), 616–627.

[31] Shweta Jain and C Seshadhri. 2017. A Fast and Provable Method for Estimating Clique Counts Using Turán's Theorem. In *Proceedings of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 441–449.

[32] Ruoming Jin, Yang Xiang, Ning Ruan, and David Fuhry. 2009. 3-HOP: a high-compression indexing scheme for reachability query. In *SIGMOD*. 813–826.

[33] Maksim Kitsak, Lazaros K Gallos, Shlomo Havlin, Fredrik Liljeros, Lev Muchnik, H Eugene Stanley, and Hernán A Makse. 2010. Identification of influential spreaders in complex networks. *Nature physics* 6, 11 (2010), 888–893.

[34] Tamara G Kolda, Ali Pinar, Todd Plantenga, C Seshadhri, and Christine Task. 2014. Counting triangles in massive graphs with MapReduce. *SIAM Journal on Scientific Computing* 36, 5 (2014), S48–S77.

[35] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *Proceedings of the 19th international conference on World wide web*. ACM, 591–600.

[36] Matthieu Latapy. 2008. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical Computer Science* 407, 1 (2008), 458–473.

[37] Victor E. Lee, Ning Ruan, Ruoming Jin, and Charu C. Aggarwal. 2010. A Survey of Algorithms for Dense Subgraph Discovery. In *Managing and Mining Graph Data*. 303–336.

[38] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data. (June 2014).

[39] Matthaios Letsios, Oana Denisa Balalau, Maximilien Danisch, Emmanuel Orsini, and Mauro Sozio. 2016. Finding heaviest k-subgraphs and events in social media. In *Data Mining Workshops (ICDMW), 2016 IEEE 16th International Conference on*. IEEE, 113–120.

[40] Kazuhisa Makino and Takeaki Uno. 2004. New algorithms for enumerating all maximal cliques. In *Algorithm Theory-SWAT 2004*. Springer, 260–272.

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

[42] Michael Mitzenmacher, Jakub Pachocki, Richard Peng, Charalampos Tsourakakis, and Shen Chen Xu. 2015. Scalable large near-clique detection in large-scale networks via sampling. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 815–824.

[43] Gergely Palla, Imre Derényi, Illés Farkas, and Tamás Vicsek. 2005. Uncovering the overlapping community structure of complex networks in nature and society. *Nature* 435, 7043 (2005), 814–818.

[44] Ali Pinar, C. Seshadhri, and Vaidyanathan Vishal. 2017. ESCAPE: Efficiently Counting All 5-Vertex Subgraphs. In *Proceedings of the 26th International Conference on World Wide Web, WWW 2017, Perth, Australia, April 3-7, 2017*. 1431–1440. https://doi.org/10.1145/3038912.3052597

[45] Angela P. Presson, Eric M. Sobel, Jeanette C. Papp, Charlyn J. Suarez, Toni Whistler, Mangalathu S. Rajeevan, Suzanne D. Vernon, and Steve Horvath. 2008. Integrated Weighted Gene Co-expression Network Analysis with an Application to Chronic Fatigue Syndrome. *BMC Systems Biology* 2 (2008), 95.

[46] Ahmet Erdem Sariyüce, C. Seshadhri, Ali Pinar, and Ümit V. Çatalyürek. 2015. Finding the Hierarchy of Dense Subgraphs using Nucleus Decompositions. In *WWW*. 927–937.

[47] Matthew C Schmidt, Nagiza F Samatova, Kevin Thomas, and Byung-Hoon Park. 2009. A scalable, parallel algorithm for maximal clique enumeration. *J. Parallel and Distrib. Comput.* 69, 4 (2009), 417–428.

[48] Kijung Shin, Tina Eliassi-Rad, and Christos Faloutsos. 2016. CoreScope: Graph Mining Using k-Core Analysis Patterns, Anomalies and Algorithms. In *Data Mining (ICDM), 2016 IEEE 16th International Conference on*. IEEE, 469–478.

[49] UNO Takeaki. 2012. Implementation issues of clique enumeration algorithm. *Special issue: Theoretical computer science and discrete mathematics, Progress in Informatics* 9 (2012), 25–30.

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

[51] Charalampos Tsourakakis. 2015. The k-clique densest subgraph problem. In *Proceedings of the 24th international conference on world wide web*. ACM, 1122–1132.

[52] Stanley Wasserman and Katherine Faust. 1994. *Social network analysis: Methods and applications*. Vol. 8. Cambridge university press.

[53] Xiao Zhou and Takao Nishizeki. 1999. Edge-Coloring and f-Coloring for Various Classes of Graphs. *MATCH Commun. Math. Comput. Chem* 51 (1999), 111–118.