A Counting-based Approach for Efficient k-Clique Densest Subgraph Discovery

Anonymous Author(s)

ABSTRACT

Densest subgraph discovery (DSD) is a fundamental topic in graph mining. It has been extensively studied in the literature and has found many real applications in a wide range of fields, such as biology, finance, and social networks. As a typical problem of DSD, the k-clique densest subgraph (CDS) problem aims to detect a subgraph from a graph, such that the ratio of the number of k-cliques over the number of its vertices is maximized. This problem has received plenty of attention in the literature, and is widely used in identifying larger "near-cliques". Existing CDS solutions, either k-core or convex programming based solutions, often need to enumerate almost all the k-cliques, which is very inefficient because real-world graphs usually have a vast number of k-cliques. To improve the efficiency, in this paper, we propose a novel framework based on the `Frank-Wolfe` algorithm, which only needs k-clique counting, rather than k-clique enumeration, where the former one is often much faster than the latter one. Based on the framework, we develop an efficient approximation algorithm, by employing the state-of-the-art k-clique counting algorithm and proposing some optimization techniques. We have performed extensive experimental evaluation on 14 real-world large graphs and the results demonstrate the high efficiency of our algorithms. Particularly, our algorithm is up to seven orders of magnitude faster than the state-of-the-art algorithm with the same accuracy guarantee.

CCS CONCEPTS

• **Theory of computation → Algorithm design techniques**.

KEYWORDS

Clique densest subgraph, convex programming, graph density

ACM Reference Format:

Anonymous Author(s). 2024. A Counting-based Approach for Efficient k-Clique Densest Subgraph Discovery. In *Proceedings of International Conference on Management of Data (SIGMOD '24).* ACM, Santiago, Chile, 18 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Densest subgraph discovery (DSD) is a fundamental topic in graph mining that has been extensively studied in recent years [3, 6, 11, 23, 44, 46, 49, 55, 57, 60]. It has found various applications in a wide range of fields, including biology [16, 25, 32, 51], finance [13, 20], and social network analysis [4, 15, 27, 28, 36, 66, 67]. The classic DSD problem [29] aims to find the subgraph with maximum edge-density, or the number of edges over the number of vertices within the subgraph, which is often called the edge-density-based densest subgraph (EDS). Recently, this problem has been generalized as the *k-clique densest subgraph* (CDS) problem [22, 23, 31, 38, 42, 47, 52, 55, 59], aiming to find the subgraph with the highest k-clique-density, which is the ratio of the number of k-cliques over the number of vertices in it. Note that since an edge can be considered as a 2-clique, the EDS problem is a special case of the CDS problem with k=2. For example, in Figure 1, the 3-clique density of the subgraph induced by $\{v_2, \cdots, v_6\}$ (in shaded region) is $\frac{7}{5}$, since there are seven 3-cliques and five vertices in it, and it is actually the 3-clique densest subgraph because there is no other subgraph with 3-clique-density larger than $\frac{7}{5}$.

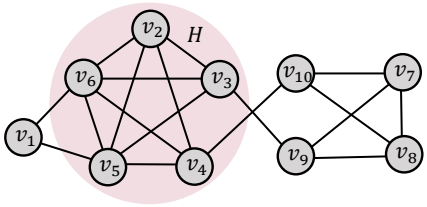


Figure 1: An example of the k-clique densest subgraph.

The CDS problem has found various real-world applications [22, 23, 31, 38, 42, 47, 55, 59]. For example, as shown in [22, 38, 42, 47, 59], the CDS can be used to detect "near-cliques", and when k gets large, it is more likely to capture useful "near-cliques", which can help discover biologically relevant functional groups [16, 35, 59, 60], find social communities [4, 12, 60], and detect anomalies [26, 56, 65]. In many of these applications, finding a "near-clique" is very important since a "near-clique" can be considered a clique in the forming stage or one with missing edges due to data corruption. For example, in protein-protein and gene-gene interaction graphs, proteins within each functional group interact with most of the rest, possibly forming a near-clique [60]; In large social networks (e.g., Facebook), large near-cliques are useful for detecting fake news [65]. Besides, finding CDS is very useful in many graph data mining applications. Specifically, it can help identify research communities in the DBLP network [23, 59, 60], detect subnetworks with a specific function in the biology network [23] and clusters in senators' networks on US bill voting [59], and discover compact dense subgraphs from e-commerce and social networks [52] when k is relatively small.

Prior works. While the CDS is very useful in practice, it is computationally costly, in both time and space, especially for large graphs. In the literature, various exact and approximation algorithms have been developed to solve the CDS problem. The exact algorithms are often based on maximum flow [29, 47, 59], k-core [23], and convex programming [19, 31, 55]. The approximation algorithms include peeling based [9, 11, 59], k-core based [23, 43], and

SIGMOD '24, June 9 - June 15, 2024, Santiago, Chile

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

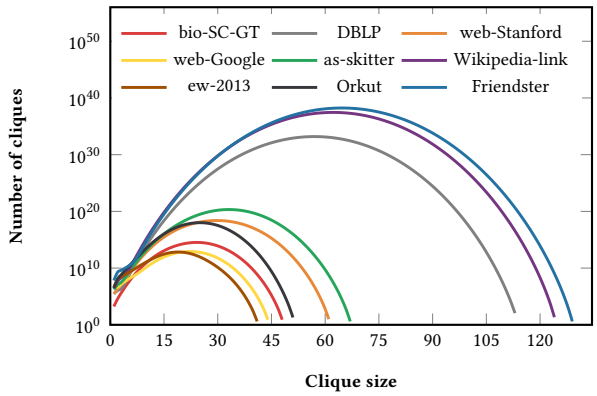


Figure 2: Trends of clique counts on nine real graphs.

convex programming based algorithms [19, 31, 55]. The state-of-the-art algorithms are KClist++ [55] and SCTL [31], both of which are based on convex programming.

The core idea of KClist++ is that the vertices present in a larger number of k-cliques are more likely to be included in the CDS. In KClist++, it first assigns a weight $r(v)$ to every vertex v in the graph, where $r(v) = 0$ initially. Then, it enumerates all the k-cliques in the graph for T iterations, and for each k-clique, it increases the minimum vertex weight in it by one. After that, KClist++ uses the vertex weights to derive a $(1+\epsilon)$-approximation solution ($\epsilon > 0$), or extracts an optimal solution from them by verifying optimality using max-flow. Here, the approximation ratio is defined as the density of CDS over that of the returned subgraph. However, KClist++ is not scalable for large k values and large-scale graphs, because, in each iteration, it needs to repeatedly enumerate all the k-cliques from the graph, which is a well-known NP-hard problem [14, 18, 34].

To alleviate the above issues, SCTL builds an index structure to speed up the k-clique enumeration process, by borrowing the succinct clique tree (SCT) from PIVOTER [34], which is the state-of-the-art algorithm for counting all k-cliques in a graph. To enumerate k-cliques, SCTL constructs an SCT to maintain a unique representation of all k-cliques, whose size is much less than the space of storing all k-cliques, allowing the SCT to be kept in the memory. By using the SCT and some optimization techniques, SCTL achieves higher efficiency than KClist++. Nonetheless, in each iteration, SCTL still needs to enumerate all the k-cliques in the worst-case to update vertex weights, making its running time proportional to the number of k-cliques in a graph.

As shown in Figure 2, the numbers of k-cliques in real-world graphs increase dramatically even for relatively small values of k. For instance, on the DBLP co-authorship network, which consists of 0.31 million of vertices and 1.04 million of edges, there are over 10^{18} 15-cliques and 10^{30} 36-cliques. Hence, enumerating almost all k-cliques is extremely costly. On the other hand, an interesting fact is that counting all the k-cliques is remarkably fast by PIVOTER [34]. For example, on the above DBLP co-authorship network, PIVOTER only requires 100ms to count all the k-cliques where k ranges from 1 to the maximum value 113, while the state-of-the-art k-clique enumeration algorithm KClist [18] needs at least two weeks to list all the k-cliques with a single $k = 9$. Hence, this motivates us to think about one natural question: *Can we find a near-optimal CDS efficiently based on k-clique counting, rather than k-clique enumeration?* In this paper, we show that it is possible to achieve this.

Table 1: Complexities of representative CDS algorithms.

Algorithm	Space complexity	Time complexity
KClist++ [55]	$O(m)$	$O(Tkm \cdot (\frac{\delta}{2})^{k-2})$
SCTL [31]	$O(n \cdot 3^{\delta/3})$	$O(n \cdot 3^{\delta/3} + Tk \cdot \lvert\Psi_k(G)\rvert)$
KCCA **(ours)**	$O(n \cdot 3^{\delta/3})$	$O(Tn \cdot 3^{\delta/3} \cdot \delta \log \delta)$

* Note: n and m are the numbers of vertices and edges in the graph G respectively, and δ denotes the degeneracy of G.
* T is the number of iterations, and $\lvert\Psi_k(G)\rvert$ denotes the number of k-cliques in G.

Our technical contributions. We propose a simple yet effective Frank-Wolfe-based framework by using k-clique counting, rather than k-clique enumeration, which is often used by existing CDS algorithms [23, 31, 55]. The Frank-Wolfe algorithm works in an iterative manner. It first assigns a weight $r(v)$ to every vertex v in the graph, which is initialized to the number of k-cliques containing v divided by k. Then, in each iteration, for each k-clique, it finds the vertex v with the minimum weight, and then updates its weight $r(v)$. By deeply analyzing the Frank-Wolfe algorithm, we find that in each iteration, the change of $r(v)$, where v is the vertex with minimum weight, can be calculated by using the number of k-cliques containing v. Based on the observation, we develop a novel framework by using a k-clique counting algorithm. The framework not only produces a near-optimal solution but also theoretically achieves a faster convergence rate than existing approaches.

Following the framework above, we employ the state-of-the-art k-clique counting algorithm PIVOTER. Besides, we propose a simultaneous weight update strategy to speed up the convergence, which assigns a more balanced distribution of weights among all vertices, allowing the algorithm to obtain the optimal solution quickly. We also design two ordering strategies to further boost efficiency. By combining the techniques above, we develop an efficient $(1+\epsilon)$ k-clique counting-based approximation (KCCA) algorithm, where $\epsilon > 0$. As shown in Table 1, a notable feature of KCCA is that its time complexity is independent of both the number of k-cliques in the graph and the value of k, so it is able to efficiently find the CDS for an arbitrary k-clique on large graphs.

Extensive experimental evaluations on 14 real-world large graphs show that KCCA achieves higher efficiency and scalability than the state-of-the-art algorithm on all datasets. Particularly, it is up to seven orders of magnitude faster than the state-of-the-art algorithm on the DBLP co-authorship network. Besides, it produces a near-optimal solution on all graphs. For instance, after 10 iterations on the Friendster graph—a social network with billions of edges—KCCA achieves an approximation ratio of 1.01. We have released the source codes and datasets of our work [1].

In summary, our main contributions are as follows.

- We propose a simple yet effective framework to break the bottleneck of existing CDS algorithms, by using k-clique

[1] https://anonymous.4open.science/r/KCCA-A410

counting, rather than k-clique enumeration. Our framework not only achieves a near-optimal solution, but also theoretically offers a faster convergence rate.
- Based on the framework above, we develop an efficient CDS algorithm by employing the state-of-the-art k-clique counting algorithm. We further propose some non-trivial optimization techniques to boost the efficiency.
- We conduct experiments on 14 real-world large graphs to demonstrate the efficiency and scalability of our algorithm.

Outline. We introduce the preliminaries in Section 2. Section 3 analyzes the limitations of state-of-the-art algorithms. We introduce our framework in Section 4, and present our KCCA algorithm in Section 5. The experimental results are reported in Section 6. We review the related works in Section 7 and conclude in Section 8.

2 PRELIMINARIES

In this section, we first present the formal definition of the CDS problem and then introduce its LP formulations.

2.1 Problem definition

Table 2: Notations and meanings.

Notation	Meaning
$G=(V,E)$	a graph with vertex set V and edge set E
$G[S]$	the subgraph of G induced by vertices in S
$\Psi_k(G)$	the set of k-cliques in G
$\Psi_k(v,G)$	the set of k-cliques containing v in G
$\mathcal{D}_k(G)$	the k-clique densest subgraph of G
$\rho_k(H)$	the k-clique density of subgraph H
$r(v)$	the weight of vertex v
α_v^C	the weight assigned to v from clique C
$\mathcal{S}_k(G)$	an approximate k-clique densest subgraph of G
Γ	a root-to-leaf path in SCT
$\mathcal{P}(\Gamma)$	the set of pivot vertices under the root-to-leaf path Γ
$\mathcal{H}(\Gamma)$	the set of hold vertices under the root-to-leaf path Γ

In this paper, we consider an unweighted and undirected graph G=(V,E), where V and E are the sets of vertices and edges in the graph, respectively. Denote by $n=|V|$ and $m=|E|$ ($m>n$) the numbers of vertices and edges in G respectively. Given a vertex set S, we use $G[S]=(S,E(S))$ to denote the subgraph of G induced by S, where $E(S)=\{(u,v)\in E \mid u,v\in S\}$ denotes the set of edges in G contained in S. For a given graph H, we also denote its sets of vertices and edges by $V(H)$ and $E(H)$, respectively.

A k-clique is a complete graph with a set C of k vertices where there is an edge between every pair of vertices. In the case without ambiguity, we simply refer to a k-clique by its set of vertices. We use $\Psi_k(G)$ to represent the set of k-cliques in G. Denote by $\Psi_k(G)=\{C\subseteq V \mid C \text{ is a } k\text{-clique of } G\}$. For each vertex $v\in G$, we use $\Psi_k(v,G)$ to denote the set of k-cliques containing v in the graph G ($k\geq 3$). We summarize the frequently used notations in Table 2.

We now formally present the definition of k-clique density.

Definition 1 (k-clique density [23, 31, 47, 55, 59]). *Given a subgraph H of a graph G and a positive integer k, the k-clique density of H, denoted by $\rho_k(H)$, is the average number of k-cliques per vertex in H, i.e.,*

$$\rho_k(H)=\frac{|\Psi_k(H)|}{|V(H)|}. \tag{1}$$

Definition 2 (k-clique densest subgraph [23, 31, 47, 55, 59]). *Given a graph G and a positive integer k, a subgraph H of G is the k-clique densest subgraph, denoted by $\mathcal{D}_k(G)$, if H has the maximum k-clique density among all subgraphs of G.*

When k=2, $\mathcal{D}_2(G)$ is the classic densest subgraph [29] that maximizes the edge-density, i.e., the average number of edges per vertex within the subgraph. In this work, we mainly focus on the cases when $k\geq 3$, and study the $(1+\epsilon)$-approximation solution ($\epsilon>0$). Here, the approximation ratio is defined as the k-clique density of CDS over that of the returned subgraph. Next, we formally present the definition of CDS problem [22, 23, 31, 38, 42, 47, 52, 55, 59].

Problem 1 (CDS problem [22, 23, 31, 38, 42, 47, 52, 55, 59]). *Given a graph G and an integer $k\geq 3$, the k-clique densest subgraph (CDS) problem aims to find the k-clique densest subgraph $\mathcal{D}_k(G)$ in G.*

Example 1. *In the graph G of Figure 1, there are ten 3-cliques, i.e., $C_1=\{v_1,v_5,v_6\}$, $C_2=\{v_2,v_4,v_5\}$, $\cdots$, $C_{10}=\{v_7,v_8,v_{10}\}$. The subgraph H of $\{v_2,v_3,v_4,v_5,v_6\}$ contains seven 3-cliques, so its 3-clique density is $\frac{7}{5}$. Clearly, H is the 3-clique densest subgraph since no other subgraph has a higher 3-clique density.*

2.2 The CP formulations of CDS problem

We first present the LP formulation of CDS problem [62]:

$$\begin{aligned}
\mathrm{LP}(G,k) \qquad & \max \sum_{C\in\Psi_k(G)} y^C \\
\text{s.t.} \qquad & \forall v\in C, y^C \leq x_v, && \forall C\in\Psi_k(G) \\
& \sum_{v\in V} x_v \leq 1, \\
& y^C\geq 0, x_v\geq 0, && \forall C\in\Psi_k(G), \forall v\in V
\end{aligned}$$

The Lagrangian dual $\mathrm{DP}(G,k)$ of the $\mathrm{LP}(G,k)$ is as follows [55]:

$$\begin{aligned}
\mathrm{DP}(G,k) \qquad & \min \max_{v\in V} r(v) \\
\text{s.t.} \qquad & r(v)=\sum_{C\in\Psi_k(v,G)} \alpha_v^C, && \forall v\in V \\
& \sum_{v\in C} \alpha_v^C = 1, && \forall C\in\Psi_k(G) \\
& \forall v\in C, \alpha_v^C\geq 0, && \forall C\in\Psi_k(G)
\end{aligned}$$

where α_v^C indicates the weight assigned to v from a clique C containing it, and $r(v)$ is the weight sum received by v from all the k-cliques containing v. Here, we introduce a new vector $\mathbf{r}$:

$$\mathbf{r}=\begin{bmatrix} r(v_1) & r(v_2) & \cdots & r(v_n)\end{bmatrix}$$

We observe that $\|\mathbf{r}\|_\infty=\max_{v\in V} r(v)$, which means that the objective function of $\mathrm{DP}(G,k)$ is: $\min\|\mathbf{r}\|_\infty$. The intuition of $\mathrm{DP}(G,k)$ is that each k-clique tries to distribute its weight, i.e., 1, to all its k vertices such that the received weights by all the vertices are as even as possible. Notice that in the CDS $\mathcal{D}_k(G)$, it is possible to distribute all cliques weights such that the weight sum received

by each vertex is exactly $\rho_k(\mathcal{D}_k(G))$, meaning that each vertex $v \in V(\mathcal{D}_k(G))$ has $r(v) = \rho_k(\mathcal{D}_k(G))$.

3 TWO STATE-OF-THE-ART ALGORITHMS

In this section, we review the two state-of-the-art CDS algorithms KClist++ [55] and SCTL [31] and further analyze their limitations.

3.1 The KClist++ algorithm

The well-known Frank-Wolfe algorithm [19, 33] can be used to solve the CP(G, k) formulation in Section 2.2, by considering a hyper-graph with the same vertices and the k-cliques as the hyper-edges [55]. A naive algorithm based on the Frank-Wolfe algorithm is presented in Algorithm 1 [55]. Specifically, for each k-clique $C = \{v_1, v_2, \cdots, v_k\}$, it maintains k weight variables $\alpha^C_{v_1}, \alpha^C_{v_2}, \cdots, \alpha^C_{v_k}$. For each vertex v, it assigns a variable $r(v)$ for storing the weight sum over all the k-cliques containing it. First, it initializes $\alpha^{C^{(0)}}_v = \frac{1}{k}$ for each vertex v in each k-clique C and $r(v) = |\Psi_k(v, G)|/k$ for each vertex v (lines 1 - 2). Then, update the weights using a for-loop

Algorithm 1: A naive Frank-Wolfe based CDS algorithm

input : A graph G and two positive integers k and T
output: An approximate CDS $\mathcal{S}_k(G)$

1 **foreach** k-clique $C \in \Psi_k(G)$ **do** $\alpha^{C^{(0)}}_v \leftarrow \frac{1}{k}, \forall v \in C$;
2 **foreach** $v \in V(G)$ **do** $r^{(0)}(v) \leftarrow |\Psi_k(v, G)|/k$;
3 **foreach** $t \leftarrow 1, 2, 3, \cdots, T$ **do**
4 **foreach** *k-clique* $C \in \Psi_k(G)$ **do**
5 $x \leftarrow \arg\min_{v \in C} r^{(t-1)}(v)$;
6 **foreach** $v \in C$ **do**
7 $\widehat{\alpha}^C_v \leftarrow 1$ if $v = x$ and 0 otherwise;
8 **foreach** *k-clique* $C \in \Psi_k(G)$ **do**
9 **foreach** $v \in C$ **do**
10 $\alpha^{C^{(t)}}_v \leftarrow (1 - \gamma_t) \cdot \alpha^{C^{(t-1)}}_v + \gamma_t \cdot \widehat{\alpha}^C_v$, with $\gamma_t = \frac{2}{t+2}$;
11 **foreach** $v \in V(G)$ **do** $r^{(t)}(v) \leftarrow \sum_{C \in \Psi_k(G): v \in C} \alpha^{C^{(t)}}_v$;
12 // Extract the *k*-clique densest subgraph
13 **foreach** $1 \le i \le |V(G)|$ **do**
14 $v_i \leftarrow$ the vertex with the i-th highest weight in $V(G)$;
15 $G_i \leftarrow$ the induced subgraph of top-i highest weight vertices;
16 $y_i \leftarrow |\Psi_k(v_i, G_i)|$;
17 $s^* \leftarrow \arg\max_{1 \le s \le n} \frac{1}{s} \sum_{i=1}^{s} y_i$;
18 **return** $\mathcal{S}_k(G) \leftarrow$ the subgraph induced by the first s^* vertices;

(lines 3-11). In each iteration, for each k-clique $C = \{v_1, v_2, \cdots, v_k\}$ in G, it first finds the vertex x with the smallest r values in C, and then updates each vertex v's $\widehat{\alpha}^C_v$, i.e., $\widehat{\alpha}^C_v$ is 1 if $v = x$, or 0 if $v \neq x$ (lines 4-7). Then, the $\alpha^{(t)}$ values of all vertices are computed as a convex combination by $\alpha^{(t-1)}$ and $\widehat{\alpha}$ (lines 8-10). Next, for each vertex v, $r(v)$ is updated as the weight sum over all the k-cliques containing it (line 11).

Intuitively, the vertices with higher weights are more likely to appear in $\mathcal{D}_k(G)$, since they are contained by more k-cliques. Thus, the subgraph induced by the first s^* vertices with the largest weights is returned as an approximate solution on G (lines 12-18). It is proved [19] that if the number of iterations T is "large enough", then the vertices with the largest r values induce an exact CDS.

The above algorithm needs to track the weight distribution to vertices of all k-cliques, requiring $O(k \cdot |\Psi_k(G)|)$ space cost. As shown in Figure 2, $|\Psi_k(G)|$ may be very large, so it is space costly. To reduce the space cost, Sun et al. [55] proposed the KClist++ algorithm by only keeping track of $r(v)$, whose space cost is $O(n)$.

Algorithm 2 presents KClist++. First, for each vertex $v \in V(G)$, it initializes $r(v) = 0$ (line 1). Then, in each iteration, it enumerates all k-cliques in G by using KClist algorithm [18] (line 3), and for each k-clique, it increases $r(v)$ by one, where v with the smallest weight in it (lines 4-6). It is noted that KClist++ will converge to the optimal solution after enumerating the k-cliques for "large enough" iterations. Even within limited iterations, it is able to yield near-optimal approximation results [55].

Limitation of KClist++. Although the space issue of directly using Frank-Wolfe algorithm has been overcome, KClist++ is still very inefficient, since it needs to enumerate all the k-cliques from scratch in each iteration, which is very time-consuming.

Algorithm 2: KClist++ [55]

input : A graph G and two positive integers k and T
output: An approximate CDS $\mathcal{S}_k(G)$

1 **foreach** $v \in V(G)$ **do** $r(v) \leftarrow 0$;
2 **foreach** $t \leftarrow 1, 2, 3, \cdots, T$ **do**
3 $\Psi_k(G) \leftarrow$ enumerate all the k-cliques in G by KClist [18];
4 **foreach** *k-clique* C *in* $\Psi_k(G)$ **do**
5 $v \leftarrow \arg\min_{u \in V(G)} r(u)$;
6 $r(v) \leftarrow r(v) + 1$;
7 $r(v) \leftarrow r(v)/T$, for each $v \in V(G)$;
8 $\mathcal{S}_k(G) \leftarrow$ run lines 13-18 of Algorithm 1;
9 **return** $\mathcal{S}_k(G)$;

3.2 The SCTL algorithm

SCTL [31] follows the same framework of KClist++, but improves it from the following three aspects:

(1) *Index-based k-clique enumeration.* To avoid enumerating the k-cliques in each iteration from scratch, SCTL designs an index by utilizing the succinct clique tree (SCT) structure from PIVOTER [34], which is a state-of-the-art algorithm for k-clique counting. Since SCT ensures a unique representation of all k-cliques, SCTL archives all k-cliques, where each root-to-leaf path represents a clique, with the path depth indicating the clique size. When enumerating k-cliques, SCTL traverses the index via the tree paths.
(2) *Reduction of the search space.* It is proved that $\mathcal{D}_k(G)$ must be located in a specific graph partition, so SCTL can process each partition individually, thereby reducing the time cost.
(3) *Batch enumeration of k-cliques.* SCTL designs an optimization technique to handle some k-cliques under the same root-to-leaf paths in a batch manner, which avoids enumerating all the k-cliques one by one.

Limitation of SCTL. A major limitation of SCTL is that in each iteration, it has to enumerate almost all the k-cliques, which is very costly. Although the enumeration process can be sped up by the index, and some k-cliques may be skipped in the batch enumeration, it is still very inefficient due to the overwhelming number

of k-cliques in real-world large graphs. For instance, as shown in Figure 2, the DBLP co-authorship network has around 6×10^{11} 7-cliques and SCTL needs over 2 hours to finish a single iteration. However, when $k = 36$, there are around 10^{30} 36-cliques, which is 10^{19} times larger than that of 7-cliques, making it impossible to find the corresponding CDS within a reasonable time cost.

4 A COUNTING-BASED CDS FRAMEWORK

As reviewed in Section 3, both the state-of-the-art algorithms SCTL and KClist++ need to enumerate almost all the k-cliques to update the vertex weights in each iteration. On the other hand, real-world graphs typically contain an exceedingly large number of k-cliques, so these algorithms require a huge amount of time to find the CDS. To break this bottleneck, in this section we propose a simple yet effective framework by using k-clique counting, rather than k-clique enumeration, to achieve a near-optimal solution. In the following, we introduce the details of our framework.

Our framework is established based on a key observation on the naive Frank-Wolfe based CDS algorithm in Section 3.1. Recall that in each iteration of Algorithm 1, for each k-clique $C \in \Psi_k(G)$, the weight of 1 unit for C is distributed to a vertex $v \in C$, where v has the minimum $r(v)$ among all vertices in C. Once all k-cliques in G have been processed in this fashion, one iteration is complete.

Particularly, in the t-th iteration, $\widehat{\alpha}_v^C$ is defined as:

$$\widehat{\alpha}_v^C = \begin{cases} 1 & \text{if } v = \arg\min_{u \in C} r^{(t-1)}(u) \\ 0 & \text{otherwise} \end{cases}$$

In this way, the updating of $r(v)$ is formulated as:

$$r^{(t)}(v) = \sum_{C \in \Psi_k(v,\, G)} \alpha_v^{C^{(t)}}$$

Based on the updating strategies above, we propose a new perspective to update $r(v)$ as follows:

$$\begin{aligned} r^{(t)}(v) &= \sum_{C \in \Psi_k(v,\, G)} \alpha_v^{C^{(t)}} \\ &= \sum_{C \in \Psi_k(v,\, G)} (1-\gamma_t) \cdot \alpha_v^{C^{(t-1)}} + \gamma_t \cdot \widehat{\alpha}_v^C \\ &= (1-\gamma_t) \cdot \sum_{C \in \Psi_k(v,\, G)} \alpha_v^{C^{(t-1)}} + \gamma_t \cdot \sum_{C \in \Psi_k(v,\, G)} \widehat{\alpha}_v^C \\ &= (1-\gamma_t) \cdot r^{(t-1)}(v) + \gamma_t \cdot \sum_{C \in \Psi_k(v,\, G)} \widehat{\alpha}_v^C \end{aligned}$$

In the context without ambiguity, we simply use $\widehat{r}(v)$ to denote the right-most term in the above equation:

$$\widehat{r}(v) = \sum_{C \in \Psi_k(v,\, G)} \widehat{\alpha}_v^C \tag{2}$$

As a result, the above perspective of updating $r(v)$ in the t-th iteration can be simplified as:

$$r^{(t)}(v) = (1-\gamma_t) \cdot r^{(t-1)}(v) + \gamma_t \cdot \widehat{r}(v) \tag{3}$$

Clearly, if we know how to obtain $\widehat{r}(v)$, then we can directly update $r^{(t)}(v)$. With a careful investigation, we find that in fact, $\widehat{r}(v)$ is exactly equal to the number of k-cliques in G that contain v, where $r^{(t-1)}(v)$ is the smallest value among all vertices in each k-clique, i.e.,

$$\widehat{r}(v) = \sum_{C \in \Psi_k(v,\, G)} \begin{cases} 1 & \text{if } v = \arg\min_{u \in C} r^{(t-1)}(u) \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

Consequently, in the t-th iteration, $\widehat{r}(v)$ can be easily derived by the following two steps: (1) identifying all vertices whose r values are less than $r^{(t-1)}(v)$ and remove them from the graph; and (2) counting all the k-cliques containing v in the reduced graph. Clearly, any arbitrary algorithm designed for counting the k-cliques containing a specific vertex, also known as local k-clique counting, can be applied to step (2). Based on the discussions above, we propose

Algorithm 3: Our proposed framework

input : A graph G and two positive integers k and T
output : An approximate CDS $\mathcal{S}_k(G)$

1 **foreach** $v \in V(G)$ **do** $r^{(0)}(v) \leftarrow |\Psi_k(v, G)|/k$;
2 **foreach** $t \leftarrow 1, 2, 3, \cdots, T$ **do**
3 sort the vertices in $V(G)$ by ascending r values ;
4 $\gamma_t \leftarrow \frac{2}{t+2}$; $G' \leftarrow G$;
5 **foreach** $v \in V(G)$ **do**
6 $\widehat{r}(v) \leftarrow$ the number of k-cliques containing v in G';
7 $r^{(t)}(v) \leftarrow (1-\gamma_t) \cdot r^{(t-1)}(v) + \gamma_t \cdot \widehat{r}(v)$;
8 remove v and its connected edges from G' ;
9 $\mathcal{S}_k(G) \leftarrow$ run lines 13-18 of Algorithm 1;
10 **return** $\mathcal{S}_k(G)$

a general clique-counting-based framework for CDS discovery, as shown in Algorithm 3, which simply replaces the shadowed codes of Algorithm 1 by the steps above. Specifically, for each vertex v, we first initialize $r(v)$ as the number of k-cliques containing v divided by k, similar to that of Algorithm 1 (line 1). Then, in each iteration, we first sort all the vertices in ascending order of their r values, set γ_t to $\frac{2}{t+2}$, and create a copy of G as G' (lines 3-4). Next, for each vertex v, we compute the number of k-cliques containing v in G', and then update $r(v)$ (lines 6-7). After that, we remove v and its connected edges from G' (line 8). Once all vertices in $V(G)$ have been processed this way, one iteration is finished. Finally, the approximate CDS can be derived using similar steps of Algorithm 1 (lines 9).

Clearly, both the time and space issues of the naive Frank-Wolfe algorithm have been addressed because we do not need $O(k \cdot |\Psi_k(G)|)$ space to keep track of the weight distribution to vertices of all k-cliques, and we also do not need to enumerate all the k-cliques. Besides, our framework follows the convergence of the Frank-Wolfe Algorithm, which is given by the following theorem:

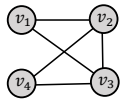
Theorem 4.1. *Suppose Δ denotes the maximum number of k-cliques that share a vertex in G. In Algorithm 3, for $t > \Omega(\frac{\Delta|\Psi_k(G)|}{\epsilon^2})$, we have $\|\mathbf{r}\|_\infty - \rho_k(\mathcal{D}_k(G)) \leq \epsilon$.*

Proof. The detailed proof is in Section A.1 of our technique report [?]. □

Hence, Algorithm 3 is guaranteed to find a $(1+\epsilon)$-approximation solution after $\Omega(\frac{\Delta|\Psi_k(G)|}{\epsilon^2})$ iterations.

Notice that the theoretical iteration number of our framework is less than $\Omega(\frac{\log(1+1/\epsilon)\Delta|\Psi_k(G)|\sqrt{k}}{\epsilon^2})$ given by KClist++ [55] and SCTL [31]. In addition, an upper bound of the optimal k-clique density can be derived in our framework based on the vertex weights similar to KClist++ in [55]. Specifically, the $\|\mathbf{r}\|_\infty$ is a decent upper bound, and tighter bounds can be derived via Lemma 13 in [55]. This is useful in estimating the approximation ratio in practice when the exact solution is unavailable.

Graph G



$C_1 = \{v_1, v_2, v_3\}$

$C_2 = \{v_2, v_3, v_4\}$

		Initialize				Iteration 1			
		v_1	v_2	v_3	v_4	v_1	v_2	v_3	v_4
$\widehat{\alpha}^{c_1}$		0	0	0	–	1	0	0	–
$\widehat{\alpha}^{c_2}$		–	0	0	0	–	0	0	1
$\widehat{r}(\cdot)$		0	0	0	0	1	0	0	1
$r(\cdot)$		1/3	2/3	2/3	1/3	7/9	2/9	2/9	7/9

Figure 3: An example for illustrating our framework.

Example 2. *In the graph G of Figure 3, there are two 3-cliques, i.e., $C_1 = \{v_1, v_2, v_3\}$ and $C_2 = \{v_2, v_3, v_4\}$. We first initialize $r(v_1) = 1/3$, $r(v_2) = 2/3$, $r(v_3) = 2/3$, and $r(v_4) = 1/3$. After running one iteration of Algorithm 3, $\widehat{\alpha}^{C_1}_{v_1}$, $\widehat{\alpha}^{C_1}_{v_2}$, $\widehat{\alpha}^{C_1}_{v_3}$, $\widehat{\alpha}^{C_2}_{v_2}$, $\widehat{\alpha}^{C_2}_{v_3}$, and $\widehat{\alpha}^{C_2}_{v_4}$ can be calculated. Afterwards, for each $v \in V(G)$, we can compute $\widehat{r}(v)$, and update $r^{(1)}(v)$ by using $\widehat{r}(v)$ and $r^{(0)}(v)$. For example, $r^{(1)}(v_1) = (1-\gamma_1) \cdot r^{(0)}(v_1) + \gamma_1 \cdot \widehat{r}(v_1) = \frac{1}{3} \cdot \frac{1}{3} + \frac{1}{3} \cdot 1 = \frac{7}{9}$.*

5 OUR KCCA ALGORITHM

In this section, we first present a basic algorithm by employing the state-of-the-art local k-clique counting algorithm PIVOTER [34] in our framework and then develop a faster optimized CDS algorithm.

5.1 A basic algorithm based on PIVOTER

The key idea of PIVOTER is that it implicitly constructs a succinct clique tree (SCT) to maintain a unique representation of all k-cliques. The SCT adapts the recursion tree of the Bron-Kerbosch algorithm for maximal clique enumeration (MCE) [58]. The Bron-Kerbosch algorithm maintains three disjoint sets $\overline{R}$, $\overline{C}$, and $\overline{X}$ in the recursive enumeration procedure, where $\overline{R}$ is a clique, $\overline{C}$ is a set of candidates that can be added to $\overline{R}$ to form a larger clique, and $\overline{X}$ is a set of vertices that have already been explored from $\overline{C}$. To compress the recursion tree, a "pivot" vertex p is selected from $\overline{C} \cup \overline{X}$ in each recursive call. In this way, any maximal clique containing $\overline{R}$ must either include p or a non-neighbor of p. Thus, the recursive calls on the neighbors of p can be skipped.

The SCT is built based on this recursion tree, where the non-neighbors of the pivot are called the "hold" vertices. By assigning each vertex a unique label, either "pivot" or "hold", each k-clique can be uniquely represented. Such a tree-shaped index has a virtual root node [2] connecting all second-level sub-trees. Each tree node stores the following information:

(1) *Vertex id*: The vertex stored in this tree node.
(2) *Vertex label*: The label of the stored vertex ("pivot" or "hold"), where the root node does not have a label.
(3) *Children*: The pointers to the child tree nodes.

Each root-to-leaf path Γ is uniquely encoded by the pivot vertices (denoted by $\mathcal{P}(\Gamma)$) and hold vertices (denoted by $\mathcal{H}(\Gamma)$) along the path [34]. In addition, we use $V(\Gamma)$ denotes all vertices in Γ, i.e., $V(\Gamma) = \mathcal{P}(\Gamma) \cup \mathcal{H}(\Gamma)$. The following lemma demonstrates how to count the number of k-cliques in each root-to-leaf path.

Lemma 5.1 ([34]). *Given a root-to-leaf path Γ, each k-clique must contain all vertices in $\mathcal{H}(\Gamma)$ and contain $k - |\mathcal{H}(\Gamma)|$ vertices in $\mathcal{P}(\Gamma)$. Each vertex in $\mathcal{P}(\Gamma)$ on this path is contained by $\binom{|\mathcal{P}(\Gamma)|-1}{k-|\mathcal{H}(\Gamma)|-1}$ k-cliques and each vertex in $\mathcal{H}(\Gamma)$ is contained by $\binom{|\mathcal{P}(\Gamma)|}{k-|\mathcal{H}(\Gamma)|}$ k-cliques.*

Example 3 illustrates how to use SCT for local k-clique counting.

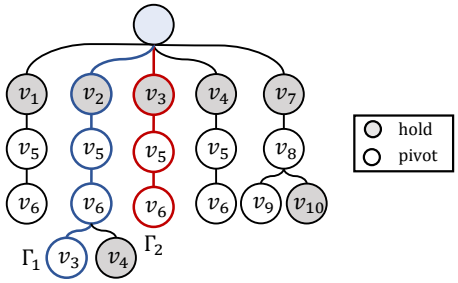


Figure 4: The SCT for the graph in Figure 1.

Example 3. *Figure 4 shows the SCT of the graph in Figure 1, where each node shows the id of the vertex it stores. For instance, to count the 3-cliques containing v_3, we need to traverse two root-to-leaf paths $\Gamma_1 = \langle root, v_2, v_5, v_6, v_3 \rangle$ and $\Gamma_2 = \langle root, v_3, v_5, v_6 \rangle$. For Γ_1, since it has one hold vertex and three pivot vertices, there are $\binom{|\mathcal{P}(\Gamma)|-1}{k-|\mathcal{H}(\Gamma)|-1} = \binom{2}{1}$ = two 3-cliques containing v_3 in Γ_1. Similarly, there is $\binom{2}{2}$ = one 3-clique containing v_3 in Γ_2. In total, there are three 3-cliques containing v_3.*

Based on the discussions above, we can adapt the SCT for the local k-clique counting in our framework. However, as stated in [31, 34], directly employing SCT for k-clique counting for a specific k would result in numerous unnecessary searches, i.e., traversing the branches that are not containing k-clique. In the wake of this, we propose three pruning criteria to avoid unnecessary searches:

(1) For a vertex v, if $cn(v) + 1 < k$, where $cn(v)$ is the core number[3] of v, then v is not contained in any k-cliques.
(2) If a branch of SCT during its construction process has $|\overline{C} \cup \overline{R}| < k$, then it does not contribute to any k-cliques.
(3) For a root-to-leaf path Γ, if it has more than k hold vertices, then it can be skipped.

Here, we briefly discuss the above pruning criteria's correctness. The first criterion holds because v must reside in the $(k-1)$-core to be contained in any k-clique. The second pruning criteria is very straightforward. For the last one, since any clique that can be counted from a path Γ must contain all its hold vertices, we can skip the path if it has more than k hold vertices when counting the k-cliques. By incorporating these three criteria, we develop a basic

[2] To avoid ambiguity, we use "node" to represent "vertex" on the SCT, and use "vertex" to represent "vertex" in the graph.

[3] The core number of v is the largest k such that there exists a k-core containing v. Here, k-core is a subgraph where each vertex has at least k neighbors in the subgraph.

CDS algorithm by employing PIVOTER, denoted by KCCA-Basic, as shown in Algorithm 4.

Algorithm 4: KCCA-Basic

input : A graph G and two positive integers k and T
output: An approximate CDS $\mathcal{S}_k(G)$

1 **foreach** $v \in V(G)$ **do** $r^{(0)}(v) \leftarrow |\Psi_k(v,G)|/k$;
2 $G \leftarrow$ locate G into a $(k-1)$-core // obtain a small graph;
3 SCT $\leftarrow$ build_SCT(G) // build the SCT for G;
4 **foreach** $t \leftarrow 1, 2, 3, \cdots, T$ **do**
5 $\quad \gamma_t \leftarrow \frac{2}{t+2}$; $\widehat{r}(v) \leftarrow 0$ for each $v \in V(G)$;
6 $\quad$ **foreach** *root-to-leaf path* $\Gamma \in SCT$ **do**
7 $\quad\quad$ **while** $\mathcal{P}(\Gamma) \neq \emptyset$ **do**
8 $\quad\quad\quad v \leftarrow \arg\min_{u \in V(\Gamma)} r^{(t-1)}(u)$;
9 $\quad\quad\quad$ **if** $v \in \mathcal{H}(\Gamma)$ **then**
10 $\quad\quad\quad\quad \widehat{r}(v) \leftarrow \widehat{r}(v) + \binom{|\mathcal{P}(\Gamma)|}{k-|\mathcal{H}(\Gamma)|}$;
11 $\quad\quad\quad\quad$ **break**;
12 $\quad\quad\quad \widehat{r}(v) \leftarrow \widehat{r}(v) + \binom{|\mathcal{P}(\Gamma)|-1}{k-|\mathcal{H}(\Gamma)|-1}$;
13 $\quad\quad\quad \mathcal{P}(\Gamma) \leftarrow \mathcal{P}(\Gamma) \backslash \{v\}$;
14 $\quad$ **foreach** $v \in V(G)$ **do**
15 $\quad\quad r^{(t)}(v) \leftarrow (1-\gamma_t) \cdot r^{(t-1)}(v) + \gamma_t \cdot \widehat{r}(v)$
16 $\mathcal{S}_k(G) \leftarrow$ run lines 13-18 of Algorithm 1;
17 **return** $\mathcal{S}_k(G)$

Similar to Algorithm 1, KCCA-Basic first initializes $r^{(0)}(v)$ for each vertex v (line 1). Then, it locates the $(k-1)$-core since the CDS must be contained by it, and builds the SCT for it (lines 2-3). Next, it uses SCT to update the weight of each vertex. Specifically, in each iteration, it first sets γ_t to $\frac{2}{t+2}$, and $\widehat{r}(v)$ to 0, which is used to record the number of k-cliques in G containing v, where v with the smallest r value among all vertices in each clique (line 5). In the t-th iteration, we traverse all the root-to-leaf paths in SCT to calculate the number of k-cliques containing each vertex v in $V(G)$, where $r^{(t-1)}(v)$ is the smallest value among all vertices in each k-clique. For each such path Γ, the number of k-cliques that include each vertex in $V(\Gamma)$ is calculated by lemma 5.1. Then, it computes $\widehat{r}(v)$ for each vertex v (lines 6-13). Once all paths in SCT are processed, it updates $r^{(t)}(v)$ for each vertex $v \in V(G)$ (lines 14-15). Finally, the CDS is extracted following similar steps of Algorithm 1.

Example 4. *Continue Example 3 where $k = 3$. Figure 5 shows the vertex weight update process in* KCCA-Basic. *The first row shows the initialized vertex weights $r^{(0)}(v)$. The second row shows the weights of $\widehat{r}(v)$ before processing the root-to-leaf path $\Gamma_1 = \langle root, v_2, v_5, v_6, v_3 \rangle$. The following rows show $\widehat{r}(v)$ when processing the path Γ_1. The seven shaded boxes contain the weights of vertices on this path, and the red boxes contain the weights of vertices that have just been increased. Since v_3 has the minimum weight among $\{v_2, v_5, v_6, v_3\}$ in the first row and $v_3 \in \mathcal{P}(\Gamma_1)$, v_3's weight is updated by the number of cliques containing it in this path, i.e., $\binom{3-1}{3-1-1} = 2$. Afterwards, v_3 is removed from the above set. Since v_2 has the minimum weight among vertices $\{v_2, v_5, v_6\}$ and $v_2 \in \mathcal{H}(\Gamma_1)$, v_2's weight is updated as $\binom{2}{3-1} = 1$.*

In the following, we prove the correctness of the Algorithm 4. Denote by $\mathcal{E}_k(\Gamma)$ the set of cliques on a root-to-leaf path Γ:

$$\mathcal{E}_k(\Gamma) = \bigcup_{\mathcal{X} \subseteq \mathcal{P}(\Gamma)} \{\mathcal{H}(\Gamma) \cup \mathcal{X}\}, \tag{5}$$

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}
weights of $r^{(0)}(\cdot)$	1/3	5/3	1	1	2	2	2/3	2/3	1/3	1/3
weights of $\hat{r}(\cdot)$	1	0	0	0	0	0	0	0	0	0
process $\{v_2, v_5, v_6, v_3\}$	1	0	2	0	0	0	0	0	0	0
process $\{v_2, v_5, v_6\}$	1	1	2	0	0	0	0	0	0	0

Figure 5: Illustrating the weight update of KCCA-Basic.

where $|\mathcal{H}(\Gamma)| + |\mathcal{X}| = k$.

We present a key lemma from [34] and use it to show the correctness.

Lemma 5.2 ([34]). *For each k-clique $C \in \Psi_k(G)$, there exist one and only one path $\Gamma \in SCT$ such that $C \in \mathcal{E}_k(\Gamma)$.*

Theorem 5.3. *Given a graph G, in t-th iteration ($t \geq 1$), Algorithm 4 correctly computes the $\widehat{r}(v)$ for each vertex $v \in V(G)$.*

Proof. W.l.o.g., suppose both Algorithms 1 and 4 break ties for vertices with the same $r(v)$ values using vertex id. Recall that $\widehat{r}(v)$ is defined in Equation (2). Based on the Lemma 5.2, $\widehat{r}(v)$ can be computed by the equation below:

$$\begin{aligned}\widehat{r}(v) &= \sum_{\Gamma \in SCT: C \in \mathcal{E}_k(\Gamma): v \in C} \widehat{\alpha}_v^C \\ &= \sum_{\Gamma \in SCT: C \in \mathcal{E}_k(\Gamma): v \in C} \begin{cases} 1 & \text{if } v = \arg\min_{u \in C} r^{(t-1)}(u) \\ 0 & \text{if otherwise} \end{cases}\end{aligned}$$

Thus, we only need to enumerate all the root-to-leaf paths containing v to obtain $\widehat{r}(v)$. Besides, for each such path Γ, if there exists a vertex with a weight smaller than $r^{(t-1)}(v)$ in $\mathcal{H}(\Gamma)$, we cannot find a k-clique C containing v where v has the smallest weight in C. If vertices in $\mathcal{P}(\Gamma)$ have weights smaller than $r^{(t-1)}(v)$, then any k-clique $C \in \mathcal{E}(\Gamma)$ containing these vertices will not contribute to $\widehat{r}(v)$. As a result, we can use the combination rule in Lemma 5.1 to calculate $\widehat{r}(v)$. The lines 6-13 in the Algorithm 4 exactly show the steps of achieving this. Hence, the theorem holds. □

Remark. KCCA-Basic is a realization of our framework (refer to Algorithm 3), where PIVOTER is only used for local k-clique counting (i.e., step (2) of our framework). It is because our framework updates a vertex v's weight $r(v)$ based on the number of k-cliques containing v. Note that any other local k-clique counting algorithms can be easily applied to our framework.

5.2 Our optimized algorithm KCCA

While KCCA-Basic is faster than KClist++ and SCTL for processing each iteration, it often requires more iterations to achieve the same approximation ratio with KClist++, as shown in our later experiments. To reduce the number of iterations, we devise a simultaneous update weight strategy coupled with two update orderings.

• **Simultaneous weight update strategy.** Most of the existing iteration-based algorithms [40, 41, 43, 53, 55] utilize a simultaneous (or asynchronous) update strategy to speed up the convergence. The key idea is that in each step, we utilize the latest updated elements of the solution to compute subsequent elements within the same iteration. For readers who are familiar with linear algebra, there is a perspective [10, 50] explaining the idea: the sequential

(synchronous) and simultaneous (asynchronous) algorithms are analogous to Jacobi and Gauss-Seidel iterations for iterative solvers, respectively. Inspired by the idea above, we develop a simultaneous weight update strategy for KCCA-Basic. Specifically, within each iteration, if a vertex v's weight $r(v)$ changes, and the updated vertex weight is promptly visible to subsequent updates of other vertices in the same iteration. The simultaneous weight update strategy enables a more balanced weight distribution among vertices, making our algorithm converge faster, as all the vertices in the densest subgraph have the same vertex weight upon convergence.

• **Update orderings.** In both KClist++ and SCTL, a more random k-clique visiting order benefits the convergence to the optimal solution [31, 55]. However, this is not always true in our framework, as shown in our experiment. To reduce the number of iterations, it is better to make the weight distribution more balanced. In light of this, we prefer to process root-to-leaf paths containing more k-cliques first, by proposing two update orderings strategies:

(1) *Depth order:* assign higher priority to root-to-leaf paths with deeper depths since they tend to contain more k-cliques.
(2) *Degeneracy order:* assign higher priority to root-to-leaf paths whose root nodes have higher degeneracy values (i.e., core numbers), since higher degeneracy values also imply more k-cliques.

By combining the above optimization techniques, we develop an optimized algorithm KCCA, as shown in Algorithm 5.

Algorithm 5: KCCA

input : A graph G and two positive integers k and T
output: An approximate CDS $\mathcal{S}_k(G)$

1 run lines 1-3 of Algorithm 4;
2 SCT ← reorder SCT by *depth order* or *degeneracy order* ;
3 **foreach** $t \leftarrow 1, 2, 3, \cdots, T$ **do**
4 $\quad \gamma_t \leftarrow \frac{2}{t+2}$;
5 $\quad$ **foreach** $v \in V(G)$ **do**
6 $\quad\quad r^{(t)}(v) \leftarrow (1-\gamma_t) \cdot r^{(t-1)}(v)$;
7 $\quad$ **foreach** *root-to-leaf path* $\Gamma \in SCT$ **do**
8 $\quad\quad$ **while** $\mathcal{P}(\Gamma) \neq \emptyset$ **do**
9 $\quad\quad\quad v \leftarrow \arg\min_{u \in V(\Gamma)} r^{(t)}(u)$;
10 $\quad\quad\quad$ **if** $v \in \mathcal{H}(\Gamma)$ **then**
11 $\quad\quad\quad\quad r^{(t)}(v) \leftarrow r^{(t)}(v) + \gamma_t \cdot \binom{|\mathcal{P}(\Gamma)|}{k-|\mathcal{H}(\Gamma)|}$;
12 $\quad\quad\quad\quad$ **break**;
13 $\quad\quad\quad r^{(t)}(v) \leftarrow r^{(t)}(v) + \gamma_t \cdot \binom{|\mathcal{P}(\Gamma)|-1}{k-|\mathcal{H}(\Gamma)|-1}$;
14 $\quad\quad\quad \mathcal{P}(\Gamma) \leftarrow \mathcal{P}(\Gamma) \backslash \{v\}$;
15 $\mathcal{S}_k(G) \leftarrow$ run lines 13-18 of Algorithm 1;
16 **return** $\mathcal{S}_k(G)$

As shown in Algorithm 5, there are two major differences between KCCA and KCCA-Basic: (1) KCCA processes the root-to-leaf paths of SCT following a fixed order (line 2), which can make it converge faster. (2) In each iteration, when a vertex v's weight $r(v)$ changes in KCCA, the updated vertex weight is immediately applied to subsequent updates within the same iteration (lines 10-13), while KCCA-Basic considers it in the next iteration.

From a theoretical perspective, KCCA needs more iterations than KCCA-Basic to achieve the same approximation ratio solution. However, in practice, KCCA always converges faster than KCCA-Basic.

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}
random order	1	3	1	2	1	0	1	1	0	0
degeneracy order	1	2	1	1	2	1	1	1	0	0
depth order	1	2	2	1	1	1	1	0	0	1

Figure 6: Illustrating different update orders.

Example 5. *Continue with Example 3 where $k = 3$. For each vertex v in the graph, since $r(v)$ can be randomly initialized, we simply set $r(v) = 0$. Then, we update vertices' weights by random order (corresponding to vertex ids), degeneracy order, and depth order, respectively, and report the results after one iteration in Figure 6, where boxes shaded in red represent the vertices selected in the solution (using lines 13-18 of Algorithm 1). Notably, the degeneracy and depth orders result in a subgraph induced by $\{v_2, v_3, v_4, v_5, v_6\}$ with density $\frac{7}{5}$ in one iteration. However, for the random order, it induces a subgraph with density 1 due to an imbalance in their assigned weights.*

To show the complexity of KCCA, we first introduce the complexity of PIVOTER.

Lemma 5.4 ([34]). *Given a graph G with n vertex and degeneracy (i.e., the maximum core number) of δ, the time and space complexity used for building SCT are both of $O(n \cdot 3^{\delta/3})$ and the SCT contains $O(n \cdot 3^{\delta/3})$ nodes.*

Lemma 5.5. *Given a graph G with degeneracy of δ, the maximum depth of the SCT of G is $O(\delta)$.*

Proof. The depth of each root-to-leaf path on the SCT is the size of the clique formed by all nodes in the path. On the other hand, the size of any clique in G cannot exceed $\delta + 1$. Hence, the lemma holds. □

Based on the above two lemmas, we can derive the following Theorem:

Theorem 5.6. *Given a graph G with n vertices and degeneracy of δ, KCCA cost $O(n \cdot 3^{\delta/3})$ space and the time complexity for each iteration is $O(n \cdot 3^{\delta/3} \cdot \delta \log \delta)$.*

Proof. The process of each path takes $O(\delta \log \delta)$ time, as each path has $O(\delta)$ nodes by Lemma 5.4. Besides, the SCT of G contains $O(n \cdot 3^{\delta/3})$ root-to-leaf paths, as it contains $O(n \cdot 3^{\delta/3})$ nodes by Lemma 5.5. Hence, the theorem holds. □

The complexity of KCCA does not depend on k, the clique size, unlike KClist++ and SCTL which scale with k. The detailed complexity comparison of the three algorithms is shown in Table 1. Note that KCCA-Basic shares the same complexity as KCCA, since they only use different update orders and strategies.

Next, we analyze the convergence rate of KCCA. Since KCCA updates its variables simultaneously and it is not a gradient-descent-like algorithm, we need analysis methods different from the one used in Section 4.

Theorem 5.7. *Suppose Δ denotes the maximum number of k-cliques that share a vertex in G. In Algorithm 5, for $t > \Omega(\frac{\Delta|\Psi_k(G)|\sqrt{k}}{\epsilon^2})$, we have $\|\mathbf{r}\|_\infty - \rho_k(\mathcal{D}_k(G)) \leq \epsilon$.*

Table 3: Approx. ratios of representative CDS algorithms.

Algorithm	approx. ratio	# iterations
KClist++ [55]	$(1+\epsilon)$	$\Omega(\frac{\log(1+1/\epsilon)\Delta\|\Psi_k(G)\|\sqrt{k}}{\epsilon^2})$
SCTL [31]	$(1+\epsilon)$	$\Omega(\frac{\log(1+1/\epsilon)\Delta\|\Psi_k(G)\|\sqrt{k}}{\epsilon^2})$
KCCA-Basic (ours)	$(1+\epsilon)$	$\Omega(\frac{\Delta\|\Psi_k(G)\|}{\epsilon^2})$
KCCA **(ours)**	$(1+\epsilon)$	$\Omega(\frac{\Delta\|\Psi_k(G)\|\sqrt{k}}{\epsilon^2})$

* Note: ϵ is the given approximation ratio, Δ is the maximum number of k-cliques that share a vertex in G, and $|\Psi_k(G)|$ denotes the number of k-cliques in G.

Hence, Algorithm 5 is guaranteed to find a $(1+\epsilon)$-approximation solution after $\Omega(\frac{\Delta|\Psi_k(G)|\sqrt{k}}{\epsilon^2})$ iterations. Besides, from a theoretical perspective, KCCA needs more iterations than KCCA-Basic to achieve the same approximation ratio solution. However, in practice, KCCA always converges faster than KCCA-Basic, since KCCA can make a more balanced vertex weight distribution. The detailed relationship between the approximation ratio and the required number of iterations in the worst case for the four algorithms is shown in table 3.

5.3 Limitations

While KCCA has achieved remarkable performance on the CDS problem, it still has some limitations. Since our algorithm employs local k-clique counting from PIVOTER, it inherits PIVOTER's limitations. A key limitation is that SCT costs $O(n \cdot \delta \cdot 3^{\delta/3})$ time for local counting, and has $O(n \cdot 3^{\delta/3})$ nodes, where n and δ are the numbers of vertex and degeneracy of the graph respectively. For the LiveJournal dataset with n=4M and δ=360, KCCA is costly in both time and space. As shown in Table 1 of [34], the local counting version cannot finish within the time limit on LiveJournal, which is much slower than global counting (6 days for global 10-clique counting [34]). Thus, the results of PIVOTER on LiveJournal in [34] suggest that the KCCA has its limits for real datasets with high degeneracy values. In addition, all CDS algorithms' complexity has an exponential relationship with δ, indicating that all algorithms (including KCCA) are not suitable for these datasets. Moreover, if any new algorithms for local k-clique counting are proposed, our framework can easily adapt these algorithms to achieve better performance.

6 EXPERIMENTS

We now present the experimental results. Section 6.1 discusses the setup. We discuss the results in Sections 6.2 and 6.3.

6.1 Setup

Datasets. We use 14 real-world datasets from different domains, which are downloaded from the Stanford Network Analysis Platform [4], Laboratory of Web Algorithmics [5], Network Repository [6], and Konect [7]. Their detailed descriptions can also be found on these websites. Table 4 reports the statistics of these graphs, where δ denotes the degeneracy of graph, and K denotes the size of the maximum clique.

[4]http://snap.stanford.edu/data/
[5]http://law.di.unimi.it/datasets.php
[6]https://networkrepository.com/network-data.php
[7]http://konect.cc/networks/

Table 4: Datasets used in our experiments.

Dataset	Category	$\|V\|$	$\|E\|$	δ	K
bio-SC-GT	Biological	1,716	31,564	60	48
econ-beacxc	Economic	507	42,176	118	87
WikiTalk	Communication	120,834	237,551	54	27
Slashdot	Comments	77,360	469,180	54	26
loc-gowalla	Locations	196,591	950,327	51	29
DBLP	Collaboration	317,080	1,049,866	113	113
web-Stanford	Web	281,903	1,992,636	71	61
web-Google	Web	916,428	4,322,051	44	44
as-skitter	Web	1,696,415	11,095,298	111	67
Wikipedia-link	Hyperlink	3,033,374	43,845,958	175	124
zhishi-baidu	Hyperlink	7,827,193	62,246,014	267	268
ew-2013	Social	4,206,785	101,355,853	145	41
Orkut	Social	3,072,627	117,185,083	253	51
Friendster	Social	124,836,180	1,806,067,135	304	129

Competitors. We evaluate the following approximation algorithms for the CDS problem:

- KCCA: our proposed algorithm in Section 5.2.
- CoreApp [23]: the core-based algorithm, which uses (h_{max}, k-clique)-core as an approximation solution of CDS, where (h, k-clique)-core is the maximal subgraph in which each vertex is contained by at least h k-cliques in the subgraph, and h_{max} is the largest h such that the (h, k-clique)-core exists. Besides, we try our best to re-implement CoreApp using C++ to make a fair comparison, which is much faster than the original implementation in Java [23].
- KClist++ [55]: the convex programming based algorithm, which is recapped in Section 3.1.
- SCTL [31]: the state-of-the-art approximation algorithm, which is discussed in Section 3.2.

Note that CoreApp [23] provides a k-approximation ratio solution, while others achieve a $(1+\epsilon)$-approximation. We follow the existing work [31] that sets the default number of iterations to 10. We implement all the algorithms in C++ and run experiments on a machine having an Intel(R) Xeon(R) Gold 6338R 2.0GHz CPU and 512GB of memory, with Ubuntu installed. If an algorithm cannot finish in two weeks, we mark its running time as **INF** in the Figure and "—" in the Table. In our experiments, we have already included the time cost of building the SCT in all results, which means that our algorithm is purely online without offline prepossessing.

Running Details. For any ϵ, the upper bound of T is calculated by Theorem 5.7. In our experiments, we follow the existing works [31, 55]: For any ϵ, each $(1+\epsilon)$-approximation algorithm (any of KCCA, KClist++, and SCTL) starts with $T = 1$ and runs for T iterations. Then, we check if the estimated error is less than ϵ. If yes, the algorithm stops; otherwise, T is set to $T \times 2$ and the process repeats until the error criterion is met. Note that the empirical value of T is usually much smaller than the upper bound.

6.2 Comparison with existing CDS algorithms

In this section, we extensively compare KCCA with KClist++ and SCTL by various experiments.

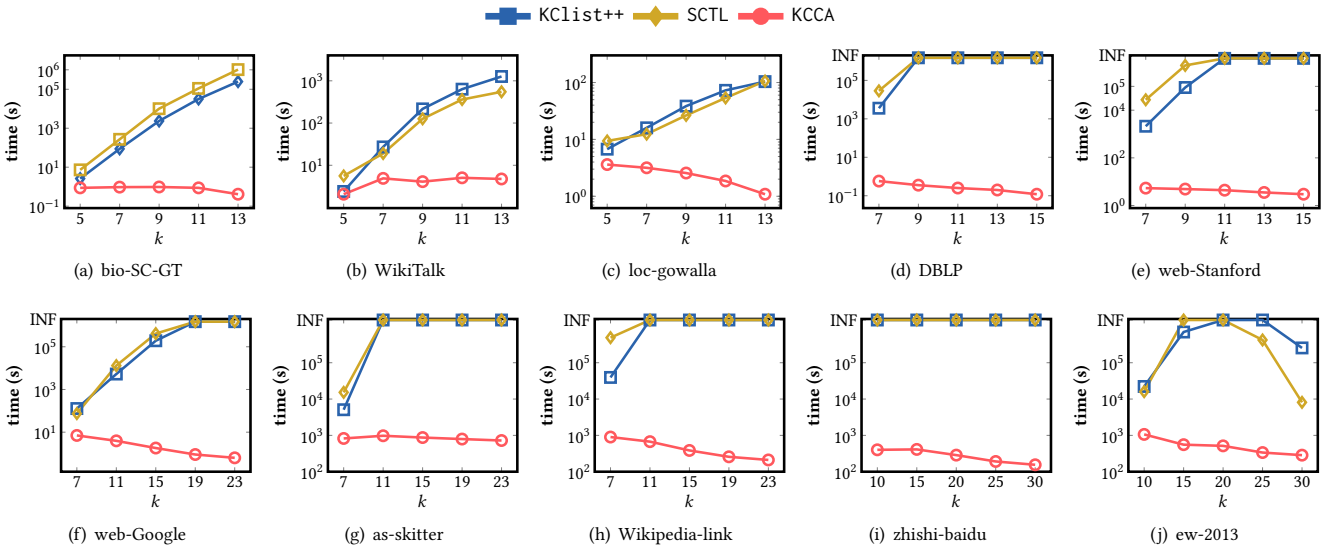


Figure 7: Effect of k on the efficiency of CoreApp, KClist++, SCTL, and KCCA.

Table 5: Efficiency on Orkut and Friendster datasets.

Dataset	k	KClist++		SCTL		KCCA	
		$1+\epsilon$	Time (s)	$1+\epsilon$	Time (s)	$1+\epsilon$	Time (s)
Orkut	15	NA	–	NA	–	< 1.01	25,310
	20	NA	–	NA	–	< 1.01	25,504
	25	NA	–	NA	–	< 1.01	15,923
	30	NA	–	NA	–	< 1.01	13,493
	35	NA	–	NA	–	< 1.01	9,939
Friendster	15	NA	–	NA	–	< 1.01	11,821
	25	NA	–	NA	–	< 1.01	7,770
	35	NA	–	NA	–	< 1.01	7,129
	45	NA	–	NA	–	< 1.01	7,240
	55	NA	–	NA	–	< 1.01	6,269
	65	NA	–	NA	–	< 1.01	6,951
	75	NA	–	NA	–	< 1.01	7,466

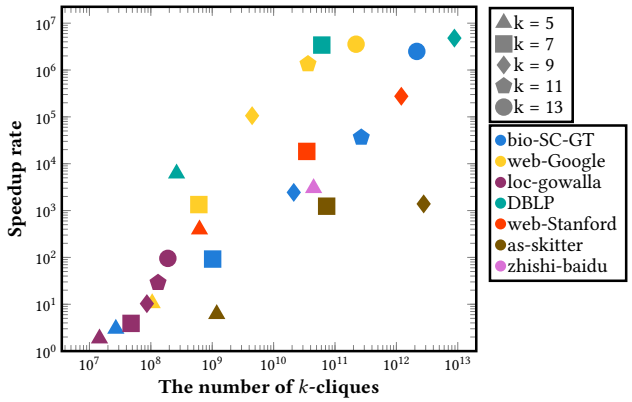


Figure 8: The number of cliques w.r.t. speedup ratio.

1. Effect of k. Figure 7 compares the average running time of these three algorithms on ten datasets by varying the clique size k, where k=5~30 and T=10. Clearly, KCCA is up to six orders of magnitude faster than KClist++ and SCTL, since it does not require enumerating the k-cliques, whereas KClist++ and SCTL struggle to enumerate a large number of k-cliques. Besides, for most of the datasets, the running time of KClist++ and SCTL increases with the growth of k, while the time cost of KCCA remains stable for different k, since its time complexity is independent of k as shown in Table 1. Moreover, it is evident that KCCA typically requires less time as k increases. This is because a larger k results in a smaller SCT (with the $(k-1)$-core decreasing in size), thus reducing the number of paths that KCCA needs to traverse.

In addition, we consider the values of k, such that the numbers of k-cliques reach the maximum values on the two largest datasets. As shown in Figure 2, when k=25 and 75, the numbers of k-cliques, i.e., 10^{18} and 10^{38}, are maximized on Orkut and Friendster datasets, respectively. Table 5 reports the running time and approximation ratios of KClist++, SCTL, and KCCA, where the numbers of iterations are set to 10. We use "NA" to denote that the algorithm could not be finished within two weeks. Clearly, by using a few thousands of seconds, KCCA can obtain solutions that are extremely close to optimal, since its approximation ratio is 1.01. However, both KClist++ and SCTL can not finish it within two weeks under all settings due to the overwhelming number of k-cliques on these datasets. To our best knowledge, KCCA is the first algorithm that can produce solutions with an approximation ratio of 1.01 for graphs with billions of edges.

2. Effect of ϵ. We evaluate the effect of ϵ using nine datasets from different domains, where each domain has a dataset, and the values of ϵ are set to 1, 0.1, 0.05, and 0.01, respectively. The experimental results are reported in Table 7, which clearly shows that KCCA outperforms the other algorithms on all datasets. Particularly, on DBLP dataset, KCCA is up to seven orders of magnitude faster than both KClist++ and SCTL That is, KCCA obtains a $(1+\epsilon)$-solution with

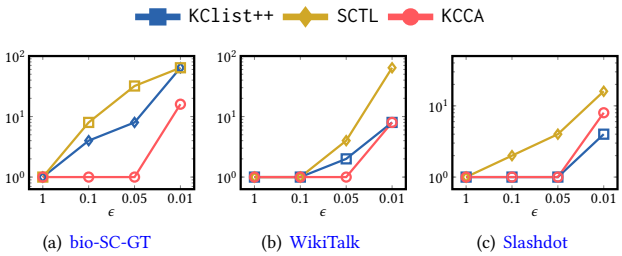
Table 6: Comparison of actual approximation ratios.

Dataset	k	CoreApp	KClist++	SCTL	KCCA
WikiTalk	5	1.399	1.001	1.052	1.006
	7	1.286	1.002	1.033	1.006
	9	1.283	1.003	1.027	1.006
	11	1.295	1.006	1.023	1.006
	13	1.179	1.013	1.013	1.009
loc-gowalla	5	1.380	1.016	1.052	1.023
	7	1.785	1.000	1.000	1.000
	9	1.541	1.000	1.000	1.000
	11	1.523	1.000	1.000	1.000
	13	1.505	1.001	1.011	1.010

$\epsilon < 0.01$ in around 100ms, while the other algorithms fail to complete it in two weeks. For around half of the datasets, KCCA is over four orders of magnitude faster than the two competitors. In addition, on the smallest datasets (bio-SC-GT and econ-beacxc), both KClist++ and SCTL struggle to produce reasonable solutions within two weeks in most cases, while KCCA takes only a few seconds and minutes to achieve solutions with $\epsilon < 0.01$ on the bio-SC-GT and econ-beaxcx datasets respectively. This is because real-world biological and economic datasets often contain a vast number of k-cliques, as shown in Figure 2. Besides, we present the number of iterations w.r.t ϵ on three datasets with $k = 11$ in Figure 9. We can observe that the empirical number of iterations required for these three algorithms shows comparable results.

3. Effect of clique numbers. We investigate the effect of the number of cliques on algorithms' performance. Specifically, we vary k from 5 to 13 with T=10, and report the speedup of KCCA over the better one from SCTL or KClist++ across seven datasets in Figure 8. We omit the results when the number of cliques is larger than 10^{13}, since SCTL or KClist++ cannot finish 10 iterations within two weeks. We observe that as the number of k-cliques increases, KCCA yields larger improvements, since it does not rely on clique enumeration, and its time complexity is independent of the number of k-cliques as shown in Table 1.

4. Actual approximation ratio. In Table 6, we report the actual approximation ratios of the approximation solutions returned by each algorithm on WikiTalk and loc-gowalla, where the numbers of iterations are set to 10. We observe that both KClist++ and kCCA yield near-optimal approximations, and they outperform CoreApp and SCTL in terms of accuracy. Hence, CoreApp performs worse than KCCA in terms of both efficiency and accuracy.



(a) bio-SC-GT (b) WikiTalk (c) Slashdot

Figure 9: The number of iterations w.r.t ϵ.

6.3 Detailed analysis of KCCA

We perform an in-depth evaluation and analysis of KCCA.

1. Time cost of different steps in KCCA. Recall that KCCA sequentially performs the following three steps: (1) reducing the original graph to $(k-1)$-core (coreReduce), (2) building the SCT (buildTree), (3) updating vertex weights with T iterations via SCT updateIter. Figure 11 shows the time cost of these three steps on ten datasets, where $k = 15$, $T = 10$, and the graphs are assumed to be loaded into memory. Note that all results reported in our paper include the time of the above three components. We see that buildTree is the most computationally expensive step on the large graphs. For example, the time cost of buildTree on ew-2013 and Friendster datasets is significantly larger than that on other datasets. Besides, on the other datasets, we observe that the third step accounts for a relatively large portion of the total time during the execution of KCCA.

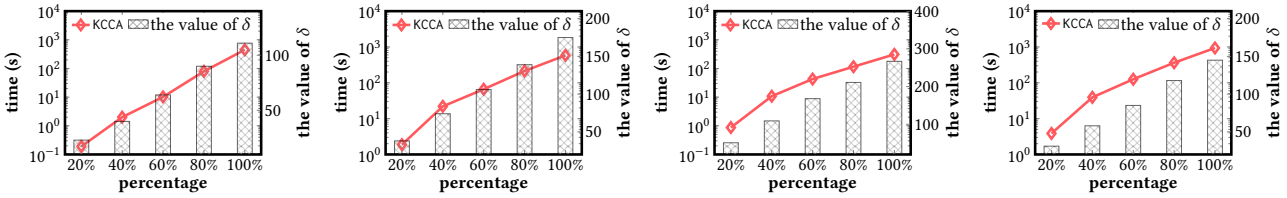
2. The impact of degeneracy on algorithm performance. In this experiment, we we evaluate the effect δ of the graph on the performance of our proposed algorithm (KCCA) with $k = 7$ and $T = 10$. Specifically, for each graph, we first randomly select 20%, 40%, 60%, 80%, and 100% of its vertices and then obtain five sub-graphs induced by these vertices, respectively. Afterwards, we run KCCA on these sub-graphs, and report both the average efficiency results and the values of δ for each sub-graph. The result is given in Figure 10. We observe that the running time of KCCA on all datasets is proportional to the value of δ, which is aligned with our previous complexity analysis in Table 1.

3. Ablation study of weight update strategies and orderings. In this experiment, we evaluate the effect of two key optimization techniques: weight update strategies and orderings. We develop six different algorithms with different update strategies and orderings to compare the variants between KCCA-Basic and KCCA. The detailed descriptions of the different variants are shown in Table 8. We then run these six algorithms on four datasets with T=10, report the approximation ratios in Table 9, and present the efficiency results on WikiTalk in Figure 15 (more details results are in our technique report []).

We make the following observations and analysis: (1) The running time of all six algorithms is almost the same, but the approximation ratios of the results of these six algorithms differ significantly. (2) The different update orderings under the sequential weight update strategy give the same result, as the vertex weight changes (i.e., sum across all k-cliques containing it) are the same w.r.t. different orders, as shown in the first three columns of the table. (3) Those algorithms that use the simultaneous weight update strategy consistently achieve lower approximation ratios than algorithms with the sequential weight update strategy, regardless of the update order. This result implies that the simultaneous weight update strategy indeed plays a crucial role in reducing the number of iterations. (4) For the later three algorithms, the random order and depth order result in comparable performance, but the degeneracy order is much better than them. Thus, we adopt the degeneracy order in our algorithms. Besides, we can infer from the results that if we want all algorithms to achieve the same empirical approximation ratios, the algorithms that perform worse in Table 9 need more iterations and time cost.

Table 7: Effect of ϵ and k. (Processing time (in seconds) of kClist++, SCTL, and KCCA; We terminate an algorithm if the upper bound of its approximation ratio is less than 1+ϵ; If an algorithm cannot finish in two weeks, we mark its running time as "—"; Best performers are highlighted in bold; We use orange, purple, green, blue, and red colors to denote the cases with 3, 4, 5, 6, and 7 orders of magnitude of improvement over the best competitor, respectively.)

Dataset	ϵ	$k=7$			$k=11$			$k=15$			$k=19$			$k=23$		
		KClist++	SCTL	KCCA	KClist++	SCTL	KCCA	KClist++	SCTL	KCCA	KClist++	SCTL	KCCA	KClist++	SCTL	KCCA
bio-SC-GT	1	9.6	25.6	**0.3**	3,532	5,581	**0.3**	343,856	609,173	**0.2**	—	—	**0.2**	—	—	**0.2**
	0.1	25.6	68.2	**0.3**	14,343	47,061	**0.3**	—	—	**0.2**	—	—	**0.2**	—	—	**0.2**
	0.05	57.8	50.4	**0.4**	27,571	126,072	**0.3**	—	—	**0.2**	—	—	**0.2**	—	—	**0.2**
	0.01	185.6	198.2	**1.3**	190,755	544,825	**1.2**	—	—	**0.7**	—	—	**0.6**	—	—	**0.6**
econ-beacxc	1	2,690	11,572	**34**	—	—	**51.6**	—	—	**51.5**	—	—	**43.1**	—	—	**51.3**
	0.1	2,690	11,572	**34**	—	—	**51.6**	—	—	**51.5**	—	—	**43.1**	—	—	**51.3**
	0.05	2,690	11,572	**34**	—	—	**51.6**	—	—	**51.5**	—	—	**43.1**	—	—	**51.3**
	0.01	2,690	356,885	**172**	—	—	**246.1**	—	—	**239.5**	—	—	**235.2**	—	—	**238.5**
WikiTalk	1	5.3	2.4	**1.4**	133	34	**1.4**	362	47	**1.3**	70	4.3	**0.7**	3.6	0.3	**0.2**
	0.1	5.3	6.6	**1.4**	133	117	**1.4**	1,001	164	**1.8**	225	7.4	**1.3**	12.5	0.5	**0.3**
	0.05	5.3	11.7	**1.4**	227	244	**1.4**	1,532	245	**2.6**	361	13.2	**2.1**	21.1	0.6	**0.5**
	0.01	10.5	81.5	**4.7**	604	992	**4.9**	4,292	835	**7.3**	1,114	89.8	**7.2**	66.5	1.3	**0.9**
Slashdot	1	4.3	3.0	**0.4**	161.8	26.7	**0.4**	453	64.3	**0.3**	126.3	6.2	**0.2**	3.4	0.8	**0.1**
	0.1	4.3	4.5	**0.4**	161.8	49.8	**0.4**	453	124.6	**0.3**	126.2	10.9	**0.3**	6.3	0.8	**0.1**
	0.05	4.3	7.4	**0.7**	161.8	95.3	**0.4**	788	245.5	**0.6**	249.8	20.2	**0.6**	10.6	1.0	**0.2**
	0.01	8.3	45.0	**2.1**	566.2	369.6	**2.0**	2,344	846.4	**1.8**	652.6	72.6	**3.2**	30.7	1.5	**0.6**
loc-gowalla	1	2.2	2.4	**1.4**	7.1	4.6	**1.1**	14.0	7.4	**0.8**	9.5	2.4	**0.6**	1.6	0.4	**0.4**
	0.1	2.2	9.1	**2.2**	7.1	4.6	**1.1**	14.0	7.4	**0.8**	9.5	2.4	**0.6**	1.6	0.4	**0.4**
	0.05	4.1	9.0	**3.0**	7.1	12.7	**1.1**	14.0	14.0	**0.9**	9.5	7.8	**0.6**	1.6	0.5	**0.4**
	0.01	11.6	29.0	**7.4**	13.8	44.0	**4.2**	27.8	105.5	**1.2**	18.6	29.1	**0.7**	4.7	1.1	**0.5**
DBLP	1	444	7,556	**0.4**	—	—	**0.3**	—	—	**0.2**	—	—	**0.1**	—	—	**0.1**
	0.1	444	28,092	**0.6**	—	—	**0.3**	—	—	**0.2**	—	—	**0.1**	—	—	**0.1**
	0.05	444	53,200	**0.7**	—	—	**0.3**	—	—	**0.2**	—	—	**0.1**	—	—	**0.1**
	0.01	444	53,200	**0.7**	—	—	**0.3**	—	—	**0.2**	—	—	**0.1**	—	—	**0.1**
web-Google	1	15.2	9.3	**3.9**	608	2,197	**2.2**	28,457	56,255	**1.3**	300,165	269,466	**0.6**	691,236	531,339	**0.4**
	0.1	28.4	73.8	**5.5**	608	8,306	**2.2**	28,457	56,255	**1.3**	300,165	269,466	**0.6**	691,236	531,339	**0.4**
	0.05	82.2	141.0	**8.5**	608	8,306	**2.2**	28,457	56,255	**1.3**	300,165	269,466	**0.6**	691,236	531,339	**0.4**
	0.01	82.2	283.9	**12.8**	1,209	16,162	**2.6**	55,654	185,611	**1.3**	539,244	—	**0.6**	—	—	**0.4**
zhishi-baidu	1	—	—	**239.4**	—	—	**210.8**	—	—	**165.7**	—	—	**128.2**	—	—	**140.4**
	0.1	—	—	**307.5**	—	—	**210.8**	—	—	**165.7**	—	—	**150.1**	—	—	**168.8**
	0.05	—	—	**307.5**	—	—	**210.8**	—	—	**165.7**	—	—	**150.1**	—	—	**168.8**
	0.01	—	—	**378.1**	—	—	**476.7**	—	—	**443.6**	—	—	**348.1**	—	—	**423.9**
ew-2013	1	624	796	**606**	11,053	5,004	**583**	143,428	90,743	**491**	593,939	275,731	**467**	614,511	132,808	**356**
	0.1	1,327	1,554	**815**	32,815	17,592	**702**	485,016	355,844	**548**	—	976,036	**552**	—	1,047,641	**424**
	0.05	2,221	2,333	**1,015**	54,955	34,398	**702**	788,963	355,845	**696**	—	—	**634**	—	—	**483**
	0.01	3,827	6,523	**3,525**	179,164	137,710	**1,598**	—	—	**1,211**	—	—	**991**	—	—	**483**



(a) as-skitter (b) Wikipedia-link (c) zhishi-baidu (d) ew-2013

Figure 10: Scalability test for KCCA algorithm.

7 RELATED WORKS

This section reviews the existing works of edge-density-based densest subgraph problem and k-clique densest subgraph problem. Other variants of the densest subgraph problem are also discussed.

• **Edge-density-based densest subgraph (EDS).** EDS aims to find the subgraph with the maximum average degree [2, 4–9, 21, 29, 30, 39, 54, 57]. This problem can be addressed by solving a parametric maximum-flow problem [29], which establishes a framework for conducting a binary search on the maximum density and using a flow network as a verification tool for EDS. In general, exact

Figure 11: Proportion of time cost of each step in KCCA.

Table 8: Descriptions of the different variants.

Name	Update Order	Update Strategy
KCCA-Basic (KB)	Random	Sequential
KCCA-BP (KBP)	Depth	Sequential
KCCA-BG (KBG)	Degeneracy	Sequential
KCCA-RA (KRA)	Random	Simultaneous
KCCA-DP (KDP)	Depth	Simultaneous
KCCA	Degeneracy	Simultaneous

Table 9: Comparison of actual approximation ratios (Red denotes the best result, and Green denotes the best result excluding KCCA).

Dataset	k	KB $1+\epsilon$	KBP $1+\epsilon$	KBG $1+\epsilon$	KRA $1+\epsilon$	KDP $1+\epsilon$	KCCA $1+\epsilon$
bio-SC-GT	5	1.967	1.967	1.967	1.037	1.058	**1.018**
	7	2.509	2.509	2.509	1.011	1.043	**1.011**
	9	2.707	2.707	2.707	1.074	1.047	**1.006**
	11	2.611	2.611	2.611	1.076	1.018	**1.009**
	13	2.429	2.429	2.429	1.084	1.061	**1.022**
WikiTalk	5	17.131	17.131	17.131	1.061	1.078	**1.006**
	7	3.114	3.114	3.114	1.052	1.067	**1.007**
	9	29.902	29.902	29.902	1.040	1.051	**1.007**
	11	27.430	27.430	27.430	1.042	1.026	**1.006**
	13	10.171	10.171	10.171	1.021	1.052	**1.009**
loc-gowalla	5	11.751	11.751	11.751	1.072	1.056	**1.024**
	7	6.423	6.423	6.423	1.059	1.058	**1.016**
	9	7.160	7.160	7.160	1.011	1.016	**1.007**
	11	4.176	4.176	4.176	1.017	1.021	**1.006**
	13	2.210	2.210	2.210	1.017	1.021	**1.006**
web-Stanford	7	3.986	3.986	3.986	1.121	1.102	**1.032**
	9	7.271	7.271	7.271	1.138	1.122	**1.040**
	11	9.525	9.525	9.525	1.148	1.126	**1.049**
	13	9.324	9.324	9.324	1.328	1.188	**1.041**
	15	11.489	11.489	11.489	1.219	1.154	**1.056**

EDS solutions are suitable for small graphs, but their performance declines for larger graphs. Consequently, researchers have turned to approximation algorithms [6, 11, 23, 37] to enhance efficiency. The peeling algorithm for k-core decomposition runs in linear time and provides a 2-approximation [11]. In addition, the EDS problem can be formulated as a convex programming and solved by the Frank-Wolfe algorithm [19, 30, 33, 55].

• **k-clique densest subgraph (CDS).** The CDS problem is proposed to better detect "near-clique" subgraphs [22, 23, 31, 42, 47, 59].

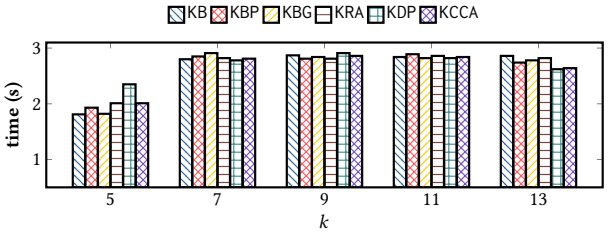


Figure 12: Efficiency of variants algorithms on WikiTalk.

Notably, when $k = 2$, this problem reduces to the well-known EDS problem. The maximum-flow based algorithm is extended to solve this problem [23, 47, 59]. Fang et al. [23] proposed a cohesive subgraph model (h, k-clique)-core for graph reduction, where (h, k-clique)-core is the maximal subgraph in which each vertex is contained by at least h k-cliques in the subgraph. Besides, they prove that the (h_{max}, k-clique)-core is a k-approximation of the CDS, where h_{max} is the largest h such that the (h, k-clique)-core exists. In addition, the convex programming based algorithms [31, 55] have been studied, which are extensively reviewed in Section 3.

• **Other variants of the densest subgraph problem.** Many variants of EDS have been studied [3, 17, 24, 44, 49, 54, 64]. The densest k-subgraph problem (*DkS*) aims to maximize the number of edges in a subgraph with k vertices, which is NP-hard [24]. Another version of EDS called optimal quasi-clique [60], extracts a subgraph, which is more compact, with a smaller diameter than the EDS. Again, this variant is NP-hard [61]. To identify locally dense regions, Qin et al.[49] proposed the top-k locally densest subgraphs problem, and Ma et al. [44] proposed a convex programming solution based on density-friendly graph decomposition [19]. In addition, [63] studies the P-mean densest subgraph problem and proposes a generalized peeling algorithm. To personalize search results, the anchored densest subgraph problem [17] aims to maximize R-subgraph density of the subgraphs containing an anchored node set. Besides, the directed densest subgraph problem is also well studied [43, 45, 46]. Recently, the fair densest subgraph problem and diverse densest subgraph problems [1, 48] have been explored to achieve equitable outcomes and overcome algorithmic bias.

8 CONCLUSIONS

In this paper, we investigate the problem of efficient k-clique densest subgraph (CDS) discovery. The existing CDS algorithms, either k-core or convex programming based solutions, often need to enumerate almost all the k-cliques, which is very inefficient because real-world graphs usually have a vast number of k-cliques. To improve the efficiency, we first propose a novel framework based on the Frank-Wolfe algorithm, which only needs k-clique counting, rather than k-clique enumeration, where the former one is often much faster than the latter one. Based on the framework, we develop an efficient approximation algorithm, by employing the state-of-the-art k-clique counting algorithm and proposing several optimization techniques. Our experimental results on 14 real-world large graphs show that our proposed algorithm is effective and efficient for the CDS problem and achieves up to seven orders of magnitude faster than the state-of-the-art algorithm. In the future, we will design distributed algorithms for the CDS problem to handle extremely large graphs that cannot be kept by a single machine.

REFERENCES

[1] Aris Anagnostopoulos, Luca Becchetti, Adriano Fazzone, Cristina Menghini, and Chris Schwiegelshohn. 2020. Spectral relaxations and fair densest subgraphs. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*. 35–44.

[2] Venkat Anantharam and Justin Salez. 2016. The densest subgraph problem in sparse random graphs. (2016).

[3] Reid Andersen and Kumar Chellapilla. 2009. Finding dense subgraphs with size bounds. In *International workshop on algorithms and models for the web-graph*. Springer, 25–37.

[4] 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. *The VLDB journal* 23 (2014), 175–199.

[5] Yuichi Asahiro, Refael Hassin, and Kazuo Iwama. 2002. Complexity of finding dense subgraphs. *Discrete Applied Mathematics* 121, 1-3 (2002), 15–26.

[6] Bahman Bahmani, Ravi Kumar, and Sergei Vassilvitskii. 2012. Densest subgraph in streaming and mapreduce. *arXiv preprint arXiv:1201.6567* (2012).

[7] Oana Denisa Balalau, Francesco Bonchi, TH Hubert Chan, Francesco Gullo, and Mauro Sozio. 2015. Finding subgraphs with maximum total density and limited overlap. In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*. 379–388.

[8] Sayan Bhattacharya, Monika Henzinger, Danupon Nanongkai, and Charalampos Tsourakakis. 2015. Space-and time-efficient algorithm for maintaining dense subgraphs on one-pass dynamic streams. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*. 173–182.

[9] Digvijay Boob, Yu Gao, Richard Peng, Saurabh Sawlani, Charalampos Tsourakakis, Di Wang, and Junxing Wang. 2020. Flowless: Extracting densest subgraphs without flow computations. In *Proceedings of The Web Conference 2020*. 573–583.

[10] Richard L. Burden and J. Douglas Faires. 2010. *Numerical Analysis* (9 ed.). Brooks/Cole, Cengage Learning.

[11] Moses Charikar. 2000. Greedy approximation algorithms for finding dense components in a graph. In *International workshop on approximation algorithms for combinatorial optimization*. Springer, 84–95.

[12] Jie Chen and Yousef Saad. 2010. Dense subgraph extraction with application to community detection. *IEEE Transactions on knowledge and data engineering* 24, 7 (2010), 1216–1230.

[13] Tianyi Chen and Charalampos Tsourakakis. 2022. Antibenford subgraphs: Unsupervised anomaly detection in financial networks. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 2762–2770.

[14] Norishige Chiba and Takao Nishizeki. 1985. Arboricity and subgraph listing algorithms. *SIAM Journal on computing* 14, 1 (1985), 210–223.

[15] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2003. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.* 32, 5 (2003), 1338–1355.

[16] Guangyu Cui, Yu Chen, De-Shuang Huang, and Kyungsook Han. 2008. An algorithm for finding functional modules and protein complexes in protein-protein interaction networks. *Journal of Biomedicine and Biotechnology* 2008 (2008).

[17] Yizhou Dai, Miao Qiao, and Lijun Chang. 2022. Anchored densest subgraph. In *Proceedings of the 2022 International Conference on Management of Data*. 1200–1213.

[18] Maximilien Danisch, Oana Balalau, and Mauro Sozio. 2018. Listing k-cliques in sparse real-world graphs. In *Proceedings of the 2018 World Wide Web Conference*. 589–598.

[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*. 233–242.

[20] 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 *Proceedings of the 15th ACM SIGKDD international conference on knowledge discovery and data mining*. 1135–1144.

[21] 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*. 300–310.

[22] Yixiang Fang, Wensheng Luo, and Chenhao Ma. 2022. Densest subgraph discovery on large graphs: Applications, challenges, and techniques. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3766–3769.

[23] Yixiang Fang, Kaiqiang Yu, Reynold Cheng, Laks VS Lakshmanan, and Xuemin Lin. 2019. Efficient algorithms for densest subgraph discovery. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1719–1732.

[24] Uriel Feige, Michael Seltser, et al. 1997. *On the densest k-subgraph problem*. Citeseer.

[25] Eugene Fratkin, Brian T Naughton, Douglas L Brutlag, and Serafim Batzoglou. 2006. MotifCut: regulatory motifs finding with maximum density subgraphs. *Bioinformatics* 22, 14 (2006), e150–e157.

[26] David Gibson, Ravi Kumar, and Andrew Tomkins. 2005. Discovering large dense subgraphs in massive graphs. In *Proceedings of the 31st international conference on Very large data bases*. 721–732.

[27] Aristides Gionis, Flavio PP Junqueira, Vincent Leroy, Marco Serafini, and Ingmar Weber. 2013. Piggybacking on social networks. In *VLDB 2013-39th International Conference on Very Large Databases*, Vol. 6. 409–420.

[28] Aristides Gionis and Charalampos E Tsourakakis. 2015. Dense subgraph discovery: Kdd 2015 tutorial. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2313–2314.

[29] Andrew V Goldberg. 1984. Finding a maximum density subgraph. (1984).

[30] Elfarouk Harb, Kent Quanrud, and Chandra Chekuri. 2022. Faster and scalable algorithms for densest subgraph and decomposition. *Advances in Neural Information Processing Systems* 35 (2022), 26966–26979.

[31] Yizhang He, Kai Wang, Wenjie Zhang, Xuemin Lin, and Ying Zhang. 2023. Scaling Up k-Clique Densest Subgraph Detection. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26.

[32] Haiyan Hu, Xifeng Yan, Yu Huang, Jiawei Han, and Xianghong Jasmine Zhou. 2005. Mining coherent dense subgraphs across massive biological networks for functional discovery. *Bioinformatics* 21, suppl_1 (2005), i213–i221.

[33] Martin Jaggi. 2013. Revisiting Frank-Wolfe: Projection-free sparse convex optimization. In *International conference on machine learning*. PMLR, 427–435.

[34] Shweta Jain and C Seshadhri. 2020. The power of pivoting for exact clique counting. In *Proceedings of the 13th International Conference on Web Search and Data Mining*. 268–276.

[35] Shweta Jain and C Seshadhri. 2020. Provably and efficiently approximating near-cliques using the Turán shadow: PEANUTS. In *Proceedings of The Web Conference 2020*. 1966–1976.

[36] Ruoming Jin, Yang Xiang, Ning Ruan, and David Fuhry. 2009. 3-hop: a high-compression indexing scheme for reachability query. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. 813–826.

[37] Samir Khuller and Barna Saha. 2009. On finding dense subgraphs. In *International colloquium on automata, languages, and programming*. Springer, 597–608.

[38] Tommaso Lanciano, Atsushi Miyauchi, Adriano Fazzone, and Francesco Bonchi. 2023. A survey on the densest subgraph problem and its variants. *arXiv preprint arXiv:2303.14467* (2023).

[39] Victor E Lee, Ning Ruan, Ruoming Jin, and Charu Aggarwal. 2010. A survey of algorithms for dense subgraph discovery. *Managing and mining graph data* (2010), 303–336.

[40] Qing Liu, Xuliang Zhu, Xin Huang, and Jianliang Xu. 2021. Local algorithms for distance-generalized core decomposition over large dynamic graphs. *Proceedings of the VLDB Endowment* 14, 9 (2021), 1531–1543.

[41] Linyuan Lü, Tao Zhou, Qian-Ming Zhang, and H Eugene Stanley. 2016. The H-index of a network node and its relation to degree and coreness. *Nature communications* 7, 1 (2016), 10168.

[42] Wensheng Luo, Chenhao Ma, Yixiang Fang, and Laks VS Lakshman. 2023. A Survey of Densest Subgraph Discovery on Large Graphs. *arXiv preprint arXiv:2306.07927* (2023).

[43] Wensheng Luo, Zhuo Tang, Yixiang Fang, Chenhao Ma, and Xu Zhou. 2023. Scalable algorithms for densest subgraph discovery. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 287–300.

[44] Chenhao Ma, Reynold Cheng, Laks VS Lakshmanan, and Xiaolin Han. 2022. Finding locally densest subgraphs: a convex programming approach. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2719–2732.

[45] Chenhao Ma, Yixiang Fang, Reynold Cheng, Laks VS Lakshmanan, and Xiaolin Han. 2022. A convex-programming approach for efficient directed densest subgraph discovery. In *Proceedings of the 2022 International Conference on Management of Data*. 845–859.

[46] Chenhao Ma, Yixiang Fang, Reynold Cheng, Laks VS Lakshmanan, Wenjie Zhang, and Xuemin Lin. 2020. Efficient algorithms for densest subgraph discovery on large directed graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1051–1066.

[47] 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*. 815–824.

[48] Atsushi Miyauchi, Tianyi Chen, Konstantinos Sotiropoulos, and Charalampos E Tsourakakis. 2023. Densest Diverse Subgraphs: How to Plan a Successful Cocktail Party with Diversity. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 1710–1721.

[49] Lu Qin, Rong-Hua Li, Lijun Chang, and Chengqi Zhang. 2015. Locally densest subgraph discovery. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 965–974.

[50] Yousef Saad. 2003. *Iterative methods for sparse linear systems*. SIAM.

[51] Barna Saha, Allison Hoch, Samir Khuller, Louiqa Raschid, and Xiao-Ning Zhang. 2010. Dense subgraphs with restrictions and applications to gene annotation graphs. In *Research in Computational Molecular Biology: 14th Annual International Conference, RECOMB 2010, Lisbon, Portugal, April 25-28, 2010. Proceedings 14*. Springer, 456–472.

[52] Raman Samusevich, Maximilien Danisch, and Mauro Sozio. 2016. Local triangle-densest subgraphs. In *2016 IEEE/ACM International Conference on Advances in*

Social Networks Analysis and Mining (ASONAM). IEEE, 33–40.

[53] Ahmet Erdem Sariyüce, C Seshadhri, and Ali Pinar. 2018. Local algorithms for hierarchical dense subgraph discovery. *Proceedings of the VLDB Endowment* 12, 1 (2018), 43–56.

[54] Saurabh Sawlani and Junxing Wang. 2020. Near-optimal fully dynamic densest subgraph. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*. 181–193.

[55] Bintao Sun, Maximilien Danisch, TH Hubert Chan, and Mauro Sozio. 2020. Kclist++: A simple algorithm for finding k-clique densest subgraphs in large graphs. *Proceedings of the VLDB Endowment (PVLDB)* (2020).

[56] Brian K Tanner, Gary Warner, Henry Stern, and Scott Olechowski. 2010. Koobface: The evolution of the social botnet. In *2010 eCrime Researchers Summit*. IEEE, 1–10.

[57] Nikolaj Tatti and Aristides Gionis. 2015. Density-friendly graph decomposition. In *Proceedings of the 24th International Conference on World Wide Web*. 1089–1099.

[58] 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.

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

[60] Charalampos Tsourakakis, Francesco Bonchi, Aristides Gionis, Francesco Gullo, and Maria Tsiarli. 2013. Denser than the densest subgraph: extracting optimal quasi-cliques with quality guarantees. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. 104–112.

[61] Charalampos E Tsourakakis. 2014. Mathematical and algorithmic analysis of network and biological data. *arXiv preprint arXiv:1407.0375* (2014).

[62] Charalampos E Tsourakakis. 2014. A novel approach to finding near-cliques: The triangle-densest subgraph problem. *arXiv preprint arXiv:1405.1477* (2014).

[63] Nate Veldt, Austin R Benson, and Jon Kleinberg. 2021. The generalized mean densest subgraph problem. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 1604–1614.

[64] Yichen Xu, Chenhao Ma, Yixiang Fang, and Zhifeng Bao. 2023. Efficient and Effective Algorithms for Generalized Densest Subgraph Discovery. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.

[65] Kaiqiang Yu and Cheng Long. 2021. Graph Mining Meets Fake News Detection. In *Data Science for Fake News: Surveys and Perspectives*. Springer, 169–189.

[66] Yang Zhang and Srinivasan Parthasarathy. 2012. Extracting analyzing and visualizing triangle k-core motifs within networks. In *2012 IEEE 28th international conference on data engineering*. IEEE, 1049–1060.

[67] Feng Zhao and Anthony KH Tung. 2012. Large scale cohesive subgraphs discovery for social network visual analysis. *Proceedings of the VLDB Endowment* 6, 2 (2012), 85–96.

A ADDITIONAL PROOFS OF LEMMAS AND THEOREMS

A.1 CONVERGENCE RATE OF OUR FRAMEWORK

To perform the convergence analysis of our framework (Algorithm 3), we only need to analyze the convergence rate of the naive Frank-Wolfe Algorithm (i.e., Algorithm 1), since our framework is a realization of Algorithm 1, which is proved in Section 4. We first formulate the convex programming of the CDS problem:

$$\mathrm{CP}(G,k) = \min\{Q_G(\alpha) \mid \alpha \in D(G,k)\} \tag{6}$$

$$D(G,k) = \{\alpha \in \mathbb{R}_+^{|\Psi_k(G)|\times k} \mid \forall C \in \Psi_k(G), \sum_{v\in C} \alpha_v^C = 1\} \tag{7}$$

$$Q_G(\alpha) = \sum_{v\in V(G)} r^2(v) \tag{8}$$

$$r(v) = \sum_{C\in\Psi_k(G)} \alpha_v^C \tag{9}$$

We summarize the Frank-Wolfe algorithm for CDS, which solves a convex program $\min Q_G(\alpha)$ subject to $\alpha \in D(G,k)$, where Q_G is twice differentiable and $D(G,k)$ is a compact convex set in some Euclidean space. In addition, we can observe that all the variants in Equations (6) - (9) share the same constraints as in $\mathrm{CP}(G,k)$, which is depicted in Section 2.2.

Here, we can use previous convergence analysis of the Frank-Wolfe algorithm [33] to shows the convergence rate of Algorithms 1 and 3. The convergence rate of these two algorithms can be described by a constant $\xi(Q_G)$ [33]:

$$\xi(Q_G) = \frac{1}{2}\mathtt{Diam}(D(G,k))^2 \sup_{\alpha\in D(G,k)} \|\nabla^2 Q_G(\alpha)\|_2, \tag{10}$$

where $\nabla^2 Q_G$ is the Hessian and $\|\cdot\|_2$ is the spectral norm of a matrix.

Lemma A.1 (Bounding $\xi(Q_G)$ [19]). *For a graph G with maximum vertex clique-degree Δ (where the maximum number of k-cliques shares a vertex in G), we have the corresponding $\xi(Q_G) \le \Delta|\Psi_k(G)|$*

The following lemma indicates that the the error in $\mathbf{r}$ can be bounded by the error in Q_G, where $\mathbf{r} = \begin{bmatrix} r(v_1) & r(v_2) & \cdots & r(v_n)\end{bmatrix}$ is the weight vector induced by given α.

Lemma A.2. *(Error in Q_G bounds error in $\mathbf{r}$) Suppose $\epsilon = \|\mathbf{r}-\mathbf{r}^*\|_2$, where $\mathbf{r}^*$ is induced by an optimal solution α^* of $\mathrm{CP}(G,k)$. Then $Q_G(\alpha) - Q_G(\alpha^*) \ge \epsilon^2$.*

Proof. The proof is as follows:

$$Q_G(\alpha) - Q_G(\alpha^*) = (\|\mathbf{r}\|_2)^2 - (\|\mathbf{r}^*\|_2)^2 \tag{11}$$

$$= (\|\mathbf{r}\|_2)^2 + (\|\mathbf{r}^*\|_2)^2 - 2\cdot(\|\mathbf{r}^*\|_2)^2 \tag{12}$$

Since $\mathbf{r}^*$ is induced by an optimal α^*, so $(\|\mathbf{r}^*\|_2)^2 \le (\|\mathbf{r}\|_2)^2$. Hence, we have:

$$(\|\mathbf{r}^*\|_2)^2 \le \|\mathbf{r}^*\|_2 \cdot \|\mathbf{r}\|_2 \tag{13}$$

Therefore, Lemma A.2 holds. □

The convergence rate of the Frank-Wolfe algorithm is:

Lemma A.3 (Convergence rate of Frank-Wolfe algorithm [33]). *Suppose $\alpha^* \in D(G,k)$ is an optimal solution of $\mathrm{CP}(G,k)$. Then for all $t \ge 1$, $Q_G(\alpha^{(t)}) - Q_G(\alpha^*) \le \frac{2\xi(Q_G)}{t+2}$.*

From the above lemmas, we can derive the convergence rate of Algorithm 3 (also Algorithm 1) by the following Theorem:

Theorem 4.1. Suppose Δ denotes the maximum number of k-cliques that share a vertex in G. In Algorithm 3, for $t > \Omega(\frac{\Delta|\Psi_k(G)|}{\epsilon^2})$, we have $\|\mathbf{r}\|_\infty - \rho_k(\mathcal{D}_k(G)) \le \epsilon$.

Proof. By using $2\Delta|\Psi_k(G)|$ as the upper bound of $\xi(Q_G)$, we claim that for Algorithm 1, if $t > \Omega(\frac{\Delta|\Psi_k(G)|}{\epsilon^2})$, then $\|\mathbf{r}-\mathbf{r}^*\|_2 \le \epsilon$. Since the density upper bound is given by $\|\mathbf{r}\|_\infty$ and $\|\mathbf{r}\|_\infty - \rho_k(\mathcal{D}_k(G)) = \|\mathbf{r}-\mathbf{r}^*\|_\infty \le \|\mathbf{r}-\mathbf{r}^*\|_2$. Hence, the Theorem 4.1 holds. □

A.2 CONVERGENCE RATE OF KCCA

Since KCCA updates its variables simultaneously and it is not a gradient-descent-like algorithm, we need analysis methods different from the one used in Section A.1. Here, we follow the definitions used in Section A.1. Notice that we do not have α in KCCA but it is still lying behind.

In the t-th iteration, $r^{(t)}(v)$ can be updated by:

$$r^{(t)}(v) = \frac{t}{t+2} r^{(t-1)}(v) + \frac{2}{t+2}\widehat{r}(v) \tag{14}$$

Rewriting Equation (14) in terms of $\alpha^{(t)}, \alpha^{(t-1)}$ and $\widehat{\alpha}$, we have:

$$\alpha^{(t)} = \frac{t}{t+2}\alpha^{(t-1)} + \frac{2}{t+2}\widehat{\alpha} \tag{15}$$

Using Equation (15), we can build a variant of Lemma 9 in [55].

Lemma A.4. *Given any $t \ge 1$, denote $\tilde{\alpha}$ by the vector that has a minimum inner product with the gradient of $Q_G(\alpha^{(t)})$*

$$\tilde{\alpha} = \begin{bmatrix}\tilde{\alpha}^{C_1} & \tilde{\alpha}^{C_2} & \cdots & \tilde{\alpha}^{C_{|\Psi_k(G)|}}\end{bmatrix} = \arg\min_{\beta\in D(G,k)} \langle \beta, \nabla Q_G(\alpha^{(t)})\rangle.$$

We have $\widehat{\alpha}$ is an approximate linear minimizer, i.e.,

$$\langle \widehat{\alpha}, \nabla Q_G(\alpha^{(t)})\rangle \le \langle \tilde{\alpha}, \nabla Q_G(\alpha^{(t)})\rangle + \frac{1}{2}\sqrt{32k}\gamma_t \xi(Q_G),$$

where $\gamma_t = \frac{2}{t+2}$, and $\xi(Q_G) = 2\Delta|\Psi_k(G)|$.

Proof. We first use $\nabla_C Q_G(\alpha)$ to denote the projection of $\nabla Q_G(\alpha)$ onto $\mathbb{R}^C$. By a straightforward calculation from proof of Lemma 4.5 [19], the (C,u)-coordinate of $\nabla Q_G(\alpha)$ is

$$2\cdot r(u) = 2\cdot \sum_{\bar{C}\in\Psi_k(G): u\in\bar{C}} \alpha_u^{\bar{C}} \tag{16}$$

It has been noted in the proof of Lemma 4.5 in [19] that one can consider each k-clique $C \in \Psi_k(G)$ independently and

$$\tilde{\alpha}^i = \arg\min_{\beta\in\mathbb{R}^{C^i}: \sum_{u\in C^i}\beta_u = 1} < \beta, \nabla_{C^i} Q(\alpha^{(t,i)}) > \tag{17}$$

Therefore, for $t \geq 1$, we have:

$$
\begin{aligned}
&< \widehat{\alpha}, \nabla Q_G(\alpha^{(t)}) > - < \bar{\alpha}, \nabla Q_G(\alpha^{(t)}) > \\
&= < \widehat{\alpha} - \bar{\alpha}, \nabla Q_G(\alpha^{(t)}) > \\
&= \sum_{i=1}^{|\Psi_k(G)|} < \widehat{\alpha}^i - \bar{\alpha}^i, \nabla_{C^i} Q_G(\alpha^{(t)}) > \\
&= \sum_{i=1}^{|\Psi_k(G)|} < \widehat{\alpha}^i - \bar{\alpha}^i, \nabla_{C^i} Q_G(\alpha^{(t,i)}) > \\
&+ \sum_{i=1}^{|\Psi_k(G)|} < \widehat{\alpha}^i - \bar{\alpha}^i, \nabla_{C^i} Q_G(\alpha^{(t)} - \alpha^{(t,i)}) > \\
&\leq \sum_{i=1}^{|\Psi_k(G)|} < \widehat{\alpha}^i - \bar{\alpha}^i, \nabla_{C^i} Q_G(\alpha^{(t)} - \alpha^{(t,i)}) > \\
&= < \widehat{\alpha} - \bar{\alpha}, \nabla Q_G(\alpha^{(t)} - (\nabla_{C^1} Q_G(\alpha^{(t,1)}, \cdots, \\
&\quad \nabla_{C^{|\Psi_k(G)|}} Q(\alpha^{(t,|\Psi_k(G)|)})) > \\
&\leq \|\widehat{\alpha} - \bar{\alpha}\| \cdot \|Q(\alpha^{(t)} - (\nabla_{C^1} Q_G(\alpha^{(t,1)}, \cdots, \\
&\quad \nabla_{C^{|\Psi_k(G)|}} Q_G(\alpha^{(t,|\Psi_k(G)|)})\|
\end{aligned}
$$

Note that $\|\widehat{\alpha} - \bar{\alpha}\| \leq \sqrt{2|\Psi_k(G)|}$ and $\frac{\partial Q_G(\alpha)}{\partial \alpha_u^C} = 2r(u)$. In the t-th iteration, $r(u)$ will be increased by at most $\frac{2\Delta}{t+2}$, so the absolute value of each coordinate of

$$Q_G(\alpha^{(t)} - (\nabla_{C^1} Q_G(\alpha^{(t,1)}, \cdots, \nabla_{C^{|\Psi_k(G)|}} Q_G(\alpha^{(t,|\Psi_k(G)|)}))))$$

is at most $\frac{2\Delta}{t+2}$. Since the dimension of α is $k \times |\Psi_k(G)|$, we have

$$\|Q_G(\alpha^{(t)} - (\nabla_{C^1} Q(\alpha^{(t,1)}, \cdots, \nabla_{C^{|\Psi_k(G)|}} Q_G(\alpha^{(t,|\Psi_k(G)|)}))))\| \leq \frac{4\Delta}{t+2}\sqrt{k|\Psi_k(G)|}$$

Therefore $< \widehat{\alpha}, \nabla Q_G(\alpha^{(t)}) > - < \bar{\alpha}, \nabla Q_G(\alpha^{(t)}) >$ can be upper bounded by:

$$\frac{4\Delta}{t+2}\sqrt{2|\Psi_k(G)|}\sqrt{k|\Psi_k(G)|} = \frac{1}{t+2}\sqrt{32k}|\Psi_k(G)|\Delta \leq \frac{1}{2}\sqrt{32k}\gamma_t \xi(Q_G)$$

□

By Lemma A.4 above, and Lemma 5 in [33], we can derive Lemma A.5, which gives the convergence of KCCA.

Lemma A.5. *For each $t \geq 1$, $\alpha^{(t)}$ computed as above satisfies $Q_G(\alpha^{(t)}) - Q_G(\alpha^*) \leq O(\frac{1}{t}) \cdot \xi(Q_G)(1 + \sqrt{32k})$, where α^* is the optimal solution to* $\mathrm{CP}(G, k)$.

Similar to Section A.1, once we have the convergence of KCCA, we can complete the analysis using the bound in ϵ and $\xi(Q_G)$

Theorem 5.5 Suppose Δ denotes the maximum number of k-cliques that share a vertex in G. In Algorithm 5, for $t > \Omega(\frac{\Delta|\Psi_k(G)|\sqrt{k}}{\epsilon^2})$, we have $\|\mathbf{r}\|_\infty - \rho_k(\mathcal{D}_k(G)) \leq \epsilon$.

Hence, Algorithm 5 is guaranteed to find a $(1+\epsilon)$-approximation solution after $\Omega(\frac{\Delta|\Psi_k(G)|\sqrt{k}}{\epsilon^2})$ iterations.

B ADDITIONAL EXPERIMENTS

B.1 The relationship between ϵ and T

To explore the relationship between ϵ and T, we present the number of iterations w.r.t. ϵ on six datasets with k = 7 and 11 in Figure 13. Note that the actual number of iterations required is consistently less than the theoretical value. For instance, on the bio-SC-GT dataset with k=11, and ϵ = 0.1, the theoretical value of T is $\Omega(10^{16})$, as calculated by Theorem 5.7, but in practice, only 1 iteration is needed for KCCA. This phenomenon also exists in other convex programming-based algorithms [31, 44, 45, 55]. In addition, the three algorithms show comparable results. Nevertheless, KCCA may require more iterations than KClist++ and SCTL, but it still takes less time as shown in Figure 14, since it avoids clique enumeration. Besides, in Table 7 of our paper, we present the comprehensive evaluated results of the running time w.r.t. different ϵ over nine datasets from the different fields.

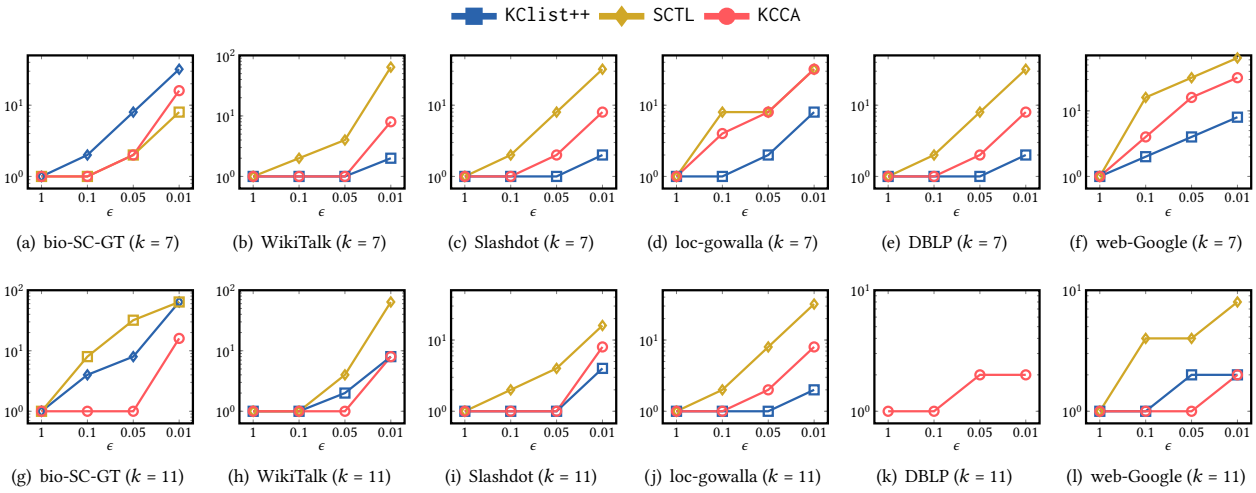
B.2 Running time of variants algorithms.

To evaluate the effect of our two key techniques, we compare variants between KCCA-Basic and KCCA. Specifically, KCCA-Basic employs a random update order and a sequential weight update strategy, whereas KCCA utilizes a degeneracy update order and a simultaneous weight update strategy. The detailed descriptions of the different variants are shown in Table 8. We then run these six algorithms on four datasets with T=10, report the approximation ratios in Table 9, and present the efficiency results in Figure 15 We can observe that the running time of all six algorithms is almost the same since the only difference between those algorithms is the order of updating vertices' weight. Besides, we can infer from the results that if we want all algorithms to achieve the same empirical approximation ratios, the algorithms that perform worse in Table 9 need more iterations and time cost.

C MORE DISCUSSIONS

The CDS solution has been used in many fundamental graph data management tasks, such as supporting graph visualization [66, 67], community detection (or search) [4, 12, 23, 59, 60], identifying near-cliques [22, 38, 42, 47, 59], and path/reachability queries [15, 36]. In addition, the CDS problem also as a key task in graph data management has gained notable attention at top-tier data management conferences, including VLDB 2019 [23], 2020 [55], 2022 [22], and SIGMOD 2023 [31].

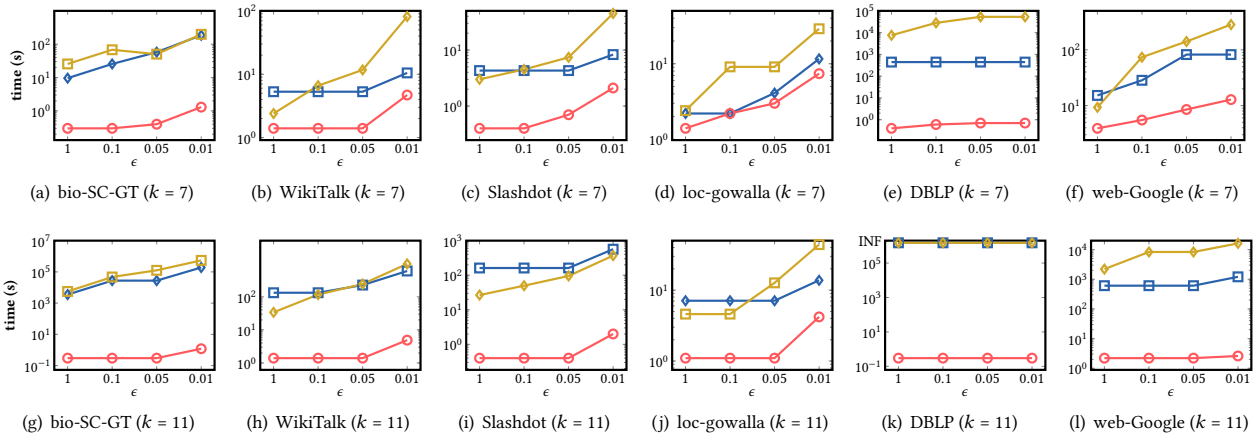
Finding CDS efficiently is very useful in many graph data mining applications. Specifically, it can help identify research communities in the DBLP network [23,59,60], detect subnetworks with a specific function in the biology network [23] and clusters in senators' networks on US bill voting [59], and discover compact dense subgraphs from e-commerce and social networks [52] when k is relatively small. In addition, identifying CDS with large k also has many applications. As shown in [31, 47, 59], a CDS becomes more akin to a large near-clique as k grows, useful for tasks like finding biologically relevant groups in protein interactions [16, 35, 59, 60], community detection [4, 12, 60], and anomaly detection [26, 56, 65], etc. In many of these applications, finding a "near-clique" is very important since a "near-clique" can be considered a clique in the forming stage or one with missing edges due to data corruption.



(a) bio-SC-GT (k = 7) (b) WikiTalk (k = 7) (c) Slashdot (k = 7) (d) loc-gowalla (k = 7) (e) DBLP (k = 7) (f) web-Google (k = 7)

(g) bio-SC-GT (k = 11) (h) WikiTalk (k = 11) (i) Slashdot (k = 11) (j) loc-gowalla (k = 11) (k) DBLP (k = 11) (l) web-Google (k = 11)

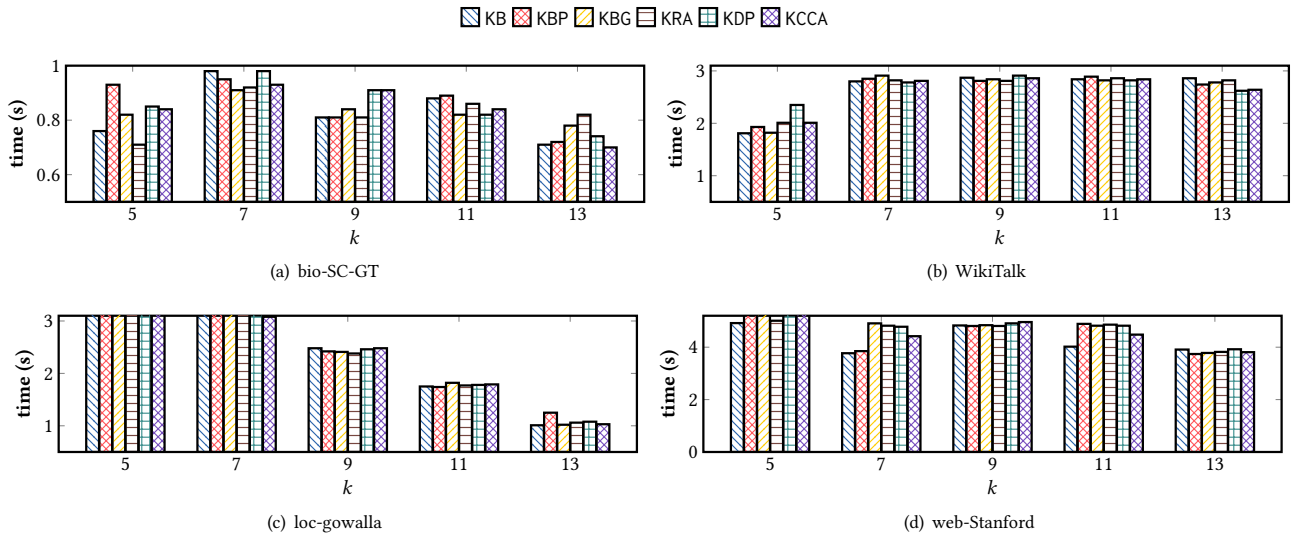
Figure 13: The number of iterations w.r.t ϵ.



(a) bio-SC-GT (k = 7) (b) WikiTalk (k = 7) (c) Slashdot (k = 7) (d) loc-gowalla (k = 7) (e) DBLP (k = 7) (f) web-Google (k = 7)

(g) bio-SC-GT (k = 11) (h) WikiTalk (k = 11) (i) Slashdot (k = 11) (j) loc-gowalla (k = 11) (k) DBLP (k = 11) (l) web-Google (k = 11)

Figure 14: Running time w.r.t ϵ.



(a) bio-SC-GT (b) WikiTalk

(c) loc-gowalla (d) web-Stanford

Figure 15: Efficiency comparison of variants algorithms.