Scaling Up k-Clique Densest Subgraph Detection

Anonymous Author(s)

ABSTRACT

In this paper, we study the k-clique densest subgraph problem, which detects the subgraph that maximizes the ratio between the number of k-cliques and the number of vertices in it. The problem has been extensively studied in the literature and has many applications in a wide range of fields such as biology and finance. Existing solutions rely heavily on repeatedly computing all the k-cliques, which are not scalable to handle large k values on large-scale graphs. In this paper, by utilizing the idea of "pivoting", we propose the SCT*-Index to compactly organize the k-cliques. Based on the SCT*-Index, our SCTL algorithm can directly obtain the k-cliques from the index and efficiently achieve near-optimal approximation. To further improve SCTL, we propose SCTL* that includes novel graph reductions and batch-processing optimizations to reduce the search space and decrease the number of visited k-cliques, respectively. As evaluated in our experiments, SCTL* significantly outperforms existing approaches by up to two orders of magnitude. In addition, we propose a sampling-based approximate algorithm that can provide reasonable approximations for any k value on billion-scale graphs. Extensive experiments on 12 real-world graphs validate both the efficiency and effectiveness of the proposed techniques.

ACM Reference Format:

Anonymous Author(s). 2018. Scaling Up k-Clique Densest Subgraph Detection. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 13 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Dense subgraph discovery is a fundamental research topic in graph mining and has been extensively studied in recent years [2, 3, 7, 34, 40, 47, 49, 51]. The study of dense subgraph discovery benefits application scenarios in a wide range of fields including biology [6, 15, 23, 29], finance [19, 28, 52], and social network analysis [3, 12, 48]. In many of these applications, finding a "near-clique" is very important since a "near-clique" can be considered as a clique in the forming stage or one with missing edges due to data corruption. For example, in a protein-protein-interaction network, an edge represents a currently known interaction between two proteins [15]. When a near-clique is detected, it is found that the missing edges are good predictions of new interactions between proteins [33, 50]. In addition, near-clique detection is also used in characterizing new proteins [15], DNA motif discovery [23], spam link detection [25] and graph compression [10]. To find such near-cliques, the k-clique densest subgraph problem is studied [22, 37, 41, 45], which aims to find the subgraph that maximizes the ratio between the number of k-cliques and the number of vertices in it. For example, in the graph shown in Figure 1, the subgraph induced by $v_2 \sim v_7$ (in grey shade) is the k-clique densest subgraph when $k = 3$. It has a k-clique density of $\frac{13}{6}$ since there are 13 k-cliques and 6 vertices in it.

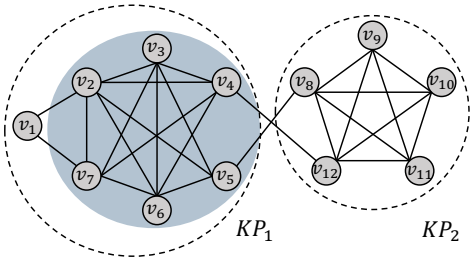


Figure 1: An example of the k-clique densest subgraph

To solve the k-clique densest subgraph problem, the classic binary-search-based approaches suffer from long running time due to a large number of max-flow calls [22, 37, 45]. Recently, another line of research focuses on convex-programming-based approaches [17, 35, 41]. Specifically, the approximate algorithm in [41] adjusts the solution in each iteration by visiting all k-cliques, which gradually converges to the optimal solution. Since it relies on computing k-cliques online for many rounds, it mainly handles k values on the lower end. In real scenarios, it has been shown that the k-clique densest subgraph is more likely to capture useful "near-cliques" when k gets large [16, 45]. Thus, it demands more efficient and scalable solutions for the k-clique densest subgraph problem.

State-of-the-arts. There are two state-of-the-art approaches [22, 41] for the k-clique densest subgraph problem in the literature.

In [22], Fang et al. propose a cohesive subgraph model (k', Ψ)-core, which is the maximum subgraph where each vertex is contained by at least k' k-cliques in the subgraph. They prove that the (k'_{max}, Ψ)-core is a $\frac{1}{k}$-approximation of the k-clique densest subgraph, where k'_{max} is the largest k' such that the (k', Ψ)-core exists. Based on this observation, they propose an algorithm to obtain the (k'_{max}, Ψ)-core as an approximate solution. In addition, they perform graph reductions using (k', Ψ)-cores and use binary search to get the exact result.

A recent work [41] formulates the k-clique densest subgraph problem as a convex program and proposes the KCL algorithm. KCL visits all the k-cliques in G for T iterations and for each visited k-clique, it increases the minimum vertex weight in it by one. After T iterations, KCL returns the top-s highest weight vertices as an approximate solution. Note that KCL is proven to converge to the optimal solution after sufficient iterations. Compared with the (k', Ψ)-core-based algorithms, KCL can yield near-optimal approximation within limited iterations since it iteratively adjusts the vertex weights to improve the current solution while the (k', Ψ)-core-based solution has a fixed approximation ratio of $\frac{1}{k}$. However,

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

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

KCL is still not scalable for large k values and large-scale graphs since (1) it needs to repeatedly compute the same k-cliques in each iteration; and (2) it unavoidably visits many k-cliques that are not contained in the optimal solution. In this paper, we focus on designing efficient and scalable convex-programming-based algorithms, with the following main challenges.

- *Challenge 1:* How to avoid computing all the k-cliques from scratch in each iteration?
- *Challenge 2:* How to reduce the search space for obtaining the k-clique densest subgraph and explore possible computation sharing opportunities when updating the vertex weights?
- *Challenge 3:* How to efficiently obtain reasonable approximations when enumerating all the k-cliques is not feasible?

Our approaches. To address Challenge 1, we adapt the succinct clique tree structure [32] and propose the SCT*-Index to organize the k-cliques compactly by utilizing the idea of "pivoting" [21, 44]. Based on the SCT*-Index, we propose the SCTL algorithm that obtains the k-cliques from the SCT*-Index instead of re-computing them from scratch as in KCL in each iteration. Moreover, SCTL only needs to traverse a small fraction of the index as k gets large. We also propose degeneracy-based and out-degree-based pre-pruning to reduce the space consumption of the index.

To address Challenge 2, we devise effective graph reductions and batch processing optimizations to speed up SCTL. Firstly, clique-connectivity-based and clique-engagement-based reductions are proposed. In clique-connectivity-based reduction, we divide the graph into k-clique-isolating partitions and consider the k-clique densest subgraph of each partition independently. In each partition, when its derived upper bound on the maximum k-clique density is dominated by the current sub-optimal solution, this partition can be safely discarded. In clique-engagement-based reduction, we observe that when a vertex is contained in only a few k-cliques w.r.t. some sub-optimal k-clique density, then it cannot be included in the optimal solution and can be discarded. In addition, we propose to handle the k-cliques and compute the vertex weight updates in a batch manner instead of processing the k-cliques individually. This is made possible by an important structural characteristic of the SCT*-Index: the paths from the root node to the leaf nodes are compact representations of k-cliques. The SCTL* algorithm with the above optimizations runs faster than existing solutions by up to two orders of magnitude as evaluated in our experiments.

To address Challenge 3, we propose a sampling-based approximate algorithm SCTL*-Sample which can provide approximate solutions even for large k values on billion-scale graphs. Its good scalability stems from the fact that it does not need to enumerate all k-cliques in any step. Initially, when sampling k-cliques, SCTL*-Sample computes the number of k-cliques needed from different parts of SCT*-Index and only visits these k-cliques. Then, we visit the sampled k-cliques for several iterations to get refined vertex weights and obtain an approximate solution on the subgraph induced by the vertices of the sampled k-cliques. Note that clique-engagement graph reduction can be applied to reduce the number of visited k-cliques. In the end, we compute the number of k-cliques of this approximate solution in the input graph by selectively traversing the SCT*-Index. In addition, we propose an exact algorithm that uses the near-optimal solution from SCTL*-Sample for graph reduction.

Contributions. Our contributions are summarized as follows:

- We adapt the succinct clique tree and propose the SCT*-Index to compactly organize k-cliques. Based on the index, the SCTL algorithm is proposed that achieves near-optimal approximation more efficiently than the state-of-the-art algorithm KCL. Note that the SCT*-Index can be built offline to support querying an arbitrary value of k.
- To further accelerate SCTL, we propose effective clique-connectivity and clique-engagement based graph reductions. In addition, we propose batch processing optimization to avoid always visiting the k-cliques individually.
- We propose a sampling-based algorithm SCTL*-Sample that provides reasonable approximate solutions even for large k values on billion-scale graphs. In addition, an exact algorithm SCTL*-Exact is proposed that prunes the search space with the solution obtained by SCTL*-Sample.
- We conduct extensive experiments on 12 real datasets to evaluate the efficiency and effectiveness of the proposed algorithms. As evaluated in the experiments, our algorithms significantly outperform the state-of-the-arts by up to two orders of magnitude, when the offline construction time of the SCT*-Index is excluded. Even for only querying a single k value and when the SCT*-Index construction time is included, our algorithms still achieve a speedup of up to one order of magnitude compared to the state-of-the-arts.

2 PRELIMINARIES

Table 1: Summary of notations

Notation	Definition
$G(V, E)$	an undirected unweighted graph
$N(v, G)$	the neighbors of v in G
$d(v, G)$	the degree of v in G
$C_k(G)$	the set of k-cliques in G
$C_k(v, G)$	the k-cliques containing vertex v in G
$\rho_k(G)$	the k-clique density of the graph G
$\mathcal{D}_k(G)$	a k-clique densest subgraph of G
$SCT(G)$	the SCT*-Index of graph G
$V_p(\mathcal{P})$	the pivot vertices under the root-to-leaf path $\mathcal{P}$
$V_h(\mathcal{P})$	the hold vertices under the root-to-leaf path $\mathcal{P}$

In this section, we present important notations and the definition of the k-clique densest subgraph problem.

2.1 Problem definition

In this paper, we consider an unweighted and undirected graph $G(V, E)$. V and E denote the set of vertices and edges in the graph, respectively. We use $n = |V|$ to denote the number of vertices and $m = |E|$ to denote the number of edges in G ($m > n$). The set of neighbors of a vertex v in G is denoted by $N(v, G)$ and the degree of v is denoted by $d(v, G) = |N(v, G)|$. Given a vertex set S, we use $G[S]$ to denote the subgraph of G induced by S. In addition, we use $C_k(G)$ to represent the set of k-cliques in G. For each vertex $v \in G$,

we use $C_k(v, G)$ to denote the set of k-cliques containing v in G ($k \geq 3$). We define the k-clique engagement of v in G as the number of k-cliques containing v in G, i.e., $|C_k(v, G)|$. Here we define the k-clique density of a graph.

Definition 1. **k-clique density.** *Given a subgraph $G' \subseteq G$ and an integer k, the k-clique density of G' (denoted by $\rho_k(G')$) is the average number of k-cliques per vertex in G', i.e., $\rho_k(G') = \frac{|C_k(G')|}{|V(G')|}$.*

Definition 2. **k-clique densest subgraph.** *Given a graph G and an integer k, a subgraph $G^* \subseteq G$ is a k-clique densest subgraph if $\rho_k(G^*) \geq \rho_k(G')$ for any $G' \subseteq G$, denoted by $\mathcal{D}_k(G)$.*

When $k = 2$, $\mathcal{D}_k(G)$ is the classic densest subgraph [26] that maximizes the average number of edges per vertex $\frac{|E(G')|}{|V(G')|}$ for any $G' \subseteq G$. In this work, we focus on the cases when $k \geq 3$.

Problem Statement. Given a graph G and an integer $k \geq 3$, the *k-clique densest subgraph* problem aims to find a k-clique densest subgraph $\mathcal{D}_k(G)$ in G.

Example 1. *Consider the graph G in Figure 1. When $k = 3$, $\mathcal{D}_k(G)$ is the subgraph induced by $v_2 \sim v_7$ with a k-clique density of $\frac{13}{6}$. When $k = 4$, $\mathcal{D}_k(G)$ is the subgraph induced by $v_2 \sim v_{12}$ with a k-clique density of 1.*

3 STATE-OF-THE-ARTS

In this section, we review two state-of-the-art approaches [22, 41] for the k-clique densest subgraph problem and discuss their limitations.

3.1 The (k', ψ)-core-based algorithms

Fang et al. [22] propose a cohesive subgraph model (k', Φ)-core and devise both approximate and exact algorithms for the k-clique densest subgraph problem based on it.

Definition 3 ([22]). **(k', Ψ)-core.** *Given a graph G, an integer k', and a k-clique instance Ψ, the (k', Ψ)-core is the subgraph $R \subseteq G$ such that (1) $|C_k(v, R)| \geq k'$ for all $v \in R$ and (2) R is maximal.*

Since (k', Ψ)-core controls the minimum k-clique engagement of vertices, its k-clique density is naturally lower-bounded by k'/k. Based on this, it is proven that the (k'_{max}, Ψ)-core itself is a $\frac{1}{k}$-approximation of the k-clique densest subgraph, where k'_{max} is the maximum k' such that (k', Ψ)-core exists. Consequently, [22] proposes the `CoreApp` algorithm that iteratively removes the vertices that are not contained by at least k' k-cliques until the (k'_{max}, Ψ)-core is found. `CoreApp` runs in $O(n\binom{d_{max}-1}{k-1})$ time and $O(m)$ space.

Aside from serving as an approximate solution, (k', Φ)-core is also used to reduce the search scope for $\mathcal{D}_k(G)$ in the exact algorithm `CoreExact` proposed in [22].

Lemma 1 ([22]). *Given a graph G and a k-clique instance Ψ, $\mathcal{D}_k(G)$ is contained in the (k', Ψ)-core, where $k' = \lceil \rho_{opt} \rceil$ and ρ_{opt} is the maximum k-clique density.*

Based on the above lemma, `CoreExact` firstly conducts (k', Φ)-core decomposition and then reduces the search scope for $\mathcal{D}_k(G)$ to some (k'', Φ)-core. During the search for $\mathcal{D}_k(G)$, we use u and l represent the tightening upper and lower bounds of the maximum k-clique density ρ_{opt}. For each connected component of the (k'', Φ)-core, a flow network is built to check if it contains a subgraph with a k-clique density at least l. If the answer is positive, it starts a binary search for ρ_{opt} on this connected component. Otherwise, this connected component is discarded. After all connected components of (k'', Φ)-core are visited, $\mathcal{D}_k(G)$ is found.

Limitations of `CoreApp` and `CoreExact`. Since `CoreExact` adopts the binary search framework, it relies on building many flow networks to verify if an approximate solution is optimal, which undermines efficiency. Although it uses (k', Ψ)-core for graph reduction, the overhead is not negligible because it needs to run the `KClist` algorithm [16] for many iterations to update the k-clique engagement of vertices when computing the (k'', Ψ)-core that contains $D_k(G)$. As for the `CoreApp` algorithm, although it can produce a result with the approximation ratio of $\frac{1}{k}$, it is hard to yield near-optimal approximation in practice since it always returns (k'_{max}, Ψ)-core, which may not highly overlap with $\mathcal{D}_k(G)$.

3.2 The convex-programming-based algorithms

Algorithm 1: `KCL`

Input: G: the input graph; T: the number of iterations
Output: $\mathcal{G}$: an approximate solution on G

1 $r(u) \leftarrow 0$ for each $u \in V(G)$;
2 **foreach** $t \leftarrow 1, 2, 3, ...T$ **do**
3 **foreach** *k-clique C in G* **do**
4 $u^* \leftarrow arg\ min_{u \in V(C)} r(u)$;
5 increase $r(u^*)$ by one;
6 $r(u) \leftarrow r(u)/T$ for each $u \in V(G)$;
7 sort the vertices in non-increasing order of $r(u)$;
8 $y_i \leftarrow$ the number of k-cliques contained in the subgraph induced by the first i vertices in the order for each $1 \leq i \leq n$;
9 $s^* \leftarrow arg\ max_{1 \leq s \leq n} \frac{1}{s} \sum_{i=1}^{s} y_i$;
10 $\mathcal{G} \leftarrow$ the subgraph induced by the first s^* vertices in the order;
11 **return** $\mathcal{G}$;

It is discovered that the edge densest subgraph problem can be formulated as a convex optimization problem and be solved via the well-known Frank-Wolfe algorithm [17, 31]. Such an algorithm is extended to solve the k-clique densest subgraph problem by considering a hyper-graph with the same vertices and the k-cliques as the hyper-edges [41]. To alleviate the memory issue from the large number of k-cliques, the `KCL` algorithm (Algorithm 1) is proposed that keeps track of the vertex weights and only requires linear memory. In `KCL`, each k-clique in G is considered to have a unit weight and distribute it among its vertices. Thus, the weight $r(u)$ of a vertex u is the total weight that u receives from the k-cliques containing it.

The main process of `KCL` is shown in Algorithm 1. Initially, $r(u)$ is set to zero for each vertex u (Line 1). In each iteration, `KCL` visits each k-clique in G. It finds the vertex u^* with the smallest weight in each visited k-clique and increases the weight of u^* by one (Lines 2-5). Intuitively, the vertices with higher weights are more likely to appear in $\mathcal{D}_k(G)$ because higher weights indicate that they are contained by many k-cliques. Thus, the subgraph induced by the first s^* vertices with the largest k-clique density is returned as an approximate solution on G (Lines 8-10).

Note that KCL adopts the KCList algorithm [16] to visit k-cliques for T iterations, each of which takes $O(km(\frac{\delta}{2})^{k-1})$ time and $O(m)$ memory (δ is the degeneracy of the graph). Then, KCL needs an additional scan of k-cliques to obtain the approximate solution. Thus, KCL takes $O((T+1)km(\frac{\delta}{2})^{k-1})$ time and $O(m)$ space in total.

The exact algorithm in [41] (denoted by KCL-Exact) runs the large-memory version of KCL (i.e., the Frank-Wolf algorithm [41]) for increasing numbers of iterations, which stores how each k-clique distributing its unit weight across vertices in the memory. KCL-Exact utilizes the concept of **stable sets** to filter out unpromising sub-optimal solutions. A vertex set B is stable if (1) the weight of any vertex in B always exceeds that of any vertex outside B; and (2) for each k-clique intersecting B but not contained by B, it must only distribute its weight among its vertices in B. Note that $D_k(G)$ must reside in a stable set [41]. Thus, KCL-Exact only checks for optimality when the current approximate solution is stable.

Limitations of KCL and KCL-Exact. Compared to previous approximate algorithms with poor approximation guarantees (e.g., CoreApp), KCL will converge to the optimal solution after visiting the k-cliques for sufficient iterations. Even within limited iterations, it is able to yield near-optimal approximation results [41]. One performance bottleneck is that KCL computes the k-cliques from scratch in each iteration in the same order. This not only compromises efficiency but also is not ideal for convergence as pointed out in [41]. Another issue is that KCL always visits all k-cliques of the input graph in each iteration, many of which are irrelevant to $D_k(G)$. As for KCL-Exact, it depends on the Frank-Wolf algorithm, which requires large memory usage and is only applicable to small graphs or small values of k.

4 UTILIZING THE IDEA OF PIVOTING

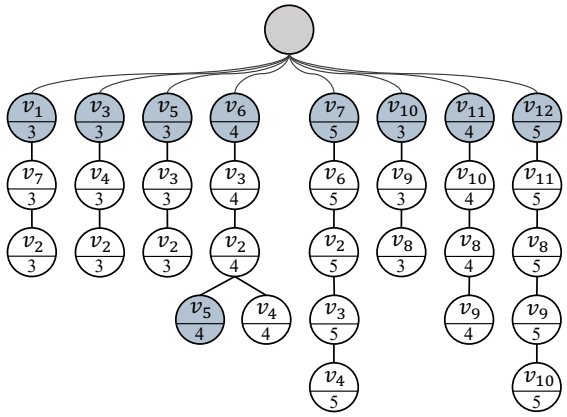


Figure 2: An example of the SCT*-Index

After reviewing the state-of-the-art solutions, we find the convex-programming-based algorithms more promising as they not only quickly yield near-optimal approximation but also return the exact solution more efficiently via a reduced number of max-flow calls. However, KCL and KCL-Exact repeatedly call KCList, which is designed to list k-cliques on sparse graphs and small values of k. In [32], adopting the idea of "pivoting" [13, 18, 21, 38, 44], the *succinct clique tree* is proposed to support exact k-clique counting for all k, which stems from the recursion tree for maximal clique enumeration. However, if we use the tree to directly support k-clique listing for a specific k, the entire tree needs to be traversed even for the paths not containing any k-clique. In this section, by adapting the succinct clique tree, we propose the SCT*-Index to help list the k-cliques without computing from scratch.

4.1 The SCT*-Index

The SCT*-Index is a tree-shaped index with a virtual root node connecting all second-level sub-trees. Each tree-node stores the following information.

- *Vertex id*: The vertex stored in this tree-node. This is empty for the root node.
- *Vertex label*: The label of the stored vertex (pivot or hold). This is empty for the root node.
- *Children*: The pointer to the child tree-nodes. This is empty for the leaf nodes.
- *Max-depth*: The maximum depth among all sub-trees that are rooted at this tree-node.

Each path $\mathcal{P}$ from the root to a leaf node is a compressed representation of k-cliques that are formed by the "piviot" vertices (denoted by $V_p(\mathcal{P})$) and "hold" vertices (denoted by $V_h(\mathcal{P})$) along the path. The vertex types (hold/pivot) are identified by the different types of recursive calls in the recursion tree [32]. When making a recursive call on a candidate set S, a pivot vertex is chosen in S (usually the vertex with the highest degree) to prune the recursive calls corresponding to the vertices in $N(p, S)$. The non-neighbors of p in S are identified are the hold vertices. We can have the following lemma as proved in [32].

Lemma 2 ([32]). *Given a root-to-leaf path $\mathcal{P}$, each k-clique in $\mathcal{P}$ must include all hold vertices and $k - |V_h(\mathcal{P})|$ pivot vertices in $\mathcal{P}$. In addition, there are $\binom{|V_p(\mathcal{P})|}{k-|V_h(\mathcal{P})|}$ k-cliques in $\mathcal{P}$ in total.*

Note that each pivot vertex does not participate in all k-cliques in $\mathcal{P}$ and is only contained by $\binom{|V_p(\mathcal{P})|-1}{k-|V_h(\mathcal{P})|-1}$ k-cliques. Obviously, only the root-to-leaf paths of lengths at least k can contain k-cliques. To enumerate k-cliques, we only explore the child tree-nodes whose max-depth is at least k.

Example 2. *Figure 2 shows the SCT*-Index of the graph in Figure 1. The top grey node represents the root node that connects the sub-trees. The shaded tree-nodes store hold vertices and the other tree-nodes store pivot vertices. Each tree-node stores a vertex id and an integer indicating the max-depth of all sub-trees rooted at this tree-node. We can see that to visit k-cliques for larger k, only a fraction of the SCT*-Index needs to be traversed. When $k = 3$, the whole SCT*-Index needs to be visited to support 3-clique listing. When $k = 4$, we only need to visit the sub-trees rooted at v_6, v_7, and v_{12}, whose max-depths are at least 4. In addition, we illustrate the conclusion in Lemma 2. In the root-to-leaf path $< root, v_6, v_3, v_2, v_5 >$, there are two hold vertices and two pivot vertices. Since all 3-cliques here must contain the two hold vertices, one additional vertex between v_3 and v_2 is needed to form 3-cliques. Thus, there are two 3-cliques under this path: $\{v_6, v_5, v_2\}$ and $\{v_6, v_5, v_3\}$. Note that for the pivot vertex v_3, it is only contained in one of them.*

Build the SCT*-Index. How do we construct the SCT*-Index? The SCT*-Index is essentially the recursion tree of listing all cliques [32]. Initially, a degeneracy ordering is used to transform G into a directed acyclic graph. For each vertex u, its out-neighborhoods $N^+(u)$ contains the neighbors of u with larger core-numbers. For each vertex u, a sub-tree rooted at u is constructed by making recursive calls on $N^+(u)$, which contains the cliques formed by u and its out-neighbors. On some large graphs, building the SCT*-Index to completion may take up too much space. In such cases, we can build a partial SCT*-Index to save space based on the following observations.

(1) for a vertex u, if $|N^+(u)+1| < k$, then the sub-tree rooted at u does not contain any k-cliques.
(2) for a vertex u, if $|cn(u)+1| < k$ ($cn(u)$ is the core-number of u), then the sub-tree rooted at u does not contain any k-cliques.

The first observation is immediate. The second observation holds because u must be contained in the $(k-1)$-core to be contained in any k-clique. Thus, we can apply out-degree-based and degeneracy-based pruning to SCT*-Index to reduce its space consumption. Specifically, we choose a threshold k' and only build the sub-trees rooted at vertices with $|N^+(u)+1| \geq k'$ and $|cn(u)+1| \geq k'$. The resulting SCT*-Index still supports k-clique listing for all $k \geq k'$, denoted by *SCT*-k'-Index*. When built to completion, the SCT*-Index takes $O(n3^{\alpha(G)/3})$ space where $\alpha(G)$ is the arboricity of graph G. The time complexity to build the SCT*-Index is $O(\alpha(G)^2|SCT(G)| + m + n)$, where $|SCT(G)|$ is the size of SCT*-Index [32].

4.2 The SCTL algorithm

Based on the SCT*-Index, we propose the SCTL algorithm that obtains near-optimal solutions more efficiently as outlined in Algorithm 2. For ease of presentation, Algorithm 2 is written in nested for-loops. In actual implementation, SCTL is a recursive algorithm that traverses the SCT*-Index in a depth-first manner to get all root-to-leaf paths by only visiting the tree-nodes whose recorded max-depths are at least k. In each iteration, SCTL visits the k-cliques in each root-to-leaf path. For each k-clique, SCTL finds the vertex with the smallest weight and increases it by 1 (Lines 2-7). Since the vertex weights are updated in the same way as in KCL, the convergence of SCTL to the optimal solution and its ability to yield near-optimal approximation is immediate based on the analysis in [41]. In addition, SCTL achieves better efficiency since it simply "reads off" the k-cliques from the root-to-leaf paths in SCT*-Index. Note that an upper bound can be derived in SCTL based on the vertex weights similar to KCL in [41]. This is useful in estimating the approximation ratio in practice when the exact solution is unavailable.

Complexity analysis. In each iteration of SCTL, the dominating time cost is incurred by traversing the SCT*-Index and updating the vertex weights. SCT*-Index traversal takes $O(\sum_{\mathcal{P}} |V(\mathcal{P})|$ times and the number of vertex weight updates equals $O(|C_k(G)|)$. At last, SCTL needs to scan the k-cliques once more to recover the approximated k-clique density (Line 8). Thus, the total time complexity of $O((T+1)\times(\sum_{\mathcal{P}} |V(\mathcal{P})|+|C_k(G)|)$. The space complexity is $|SCT(G)|$.

Algorithm 2: SCTL

Input: G: the input graph; T: the number of iterations, $SCT(G)$: the SCT-index of G
Output: $\mathcal{G}$: an approximate solution on G

1 $r(u) \leftarrow 0$ for all $u \in V(G)$;
2 **foreach** $t \leftarrow 1, 2, 3, ...T$ **do**
3 **foreach** *valid root-to-leaf path* $\mathcal{P}$ **do**
4 collect the pivot vertices and hold vertices in $\mathcal{P}$;
5 **foreach** *k-clique* C *in* $\mathcal{P}$ **do**
6 $u^* \leftarrow arg\ min_{u \in V(C)} r(u)$;
7 Increase $r(u^*)$ by 1;
8 run Lines 6-10 of Algorithm 1;
9 **return** $\mathcal{G}$;

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}
weights after 1st iteration	1	0	1	2	1	3	6	0	0	1	3	6
process clique $\{v_6, v_5, v_3\}$	1	0	2	2	1	3	6	0	0	1	3	6
process clique $\{v_6, v_5, v_2\}$	1	1	2	2	1	3	6	0	0	1	3	6

Figure 3: The weight updates in SCTL

Example 3. *We show how* SCTL *updates vertex weights in the* 2^{nd} *iteration when* $k = 3$. *In Figure 3, the vertex weights after the* 1^{st} *iteration of* SCTL *are shown in the first row. The following rows show the vertex weights after processing the* 3*-cliques in the root-to-leaf path* $<root, v_6, v_3, v_2, v_5>$. *The shaded boxes contain the vertex weights of the clique being visited and red boxes contain the vertex weight that has just been increased. Before clique* $\{v_6, v_5, v_3\}$ *is visited,* v_3 *has the minimum weight and its weight is increase to* 2. *When processing clique* $\{v_6, v_5, v_2\}$, *the weight of* v_3 *is increased from* 0 *to* 1.

5 OPTIMIZATIONS

Based on the SCT*-Index, the SCTL algorithm can list k-cliques and update vertex weights much faster than KClist. To further improve efficiency, we devise effective graph reduction rules to limit the search scope for the optimal solution by considering clique connectivity and clique engagement. In addition, we design batch processing techniques to reduce the total number of vertex weight updates in each iteration based on the structure of SCT*-Index.

5.1 Graph reductions

We observe that neither of KCL and SCTL exploits the pruning power of sub-optimal solutions and always visits all the k-cliques in G in each iteration. Motivated by this, we present the k-clique-connectivity-based and the clique-engagement-based reductions.

Clique-connectivity-based reduction. Here we aim to obtain a graph partitioning of G and apply "divide-and-conquer" to each partition. The connected components of G are a natural fit for this task but such a partition is too coarse for the k-clique densest subgraph problem. Intuitively, when two subgraphs induced by two vertex partitions do not share any k-cliques, they can be considered individually for the k-clique densest subgraph problem. Thus, we define the k-clique-isolating partition as follows, which divides the graph into more fine-grained partitions.

Definition 4. **k-clique-isolating partition.** *Given a graph G and an integer k, a k-clique-isolating partition of $|V(G)|$ is a set of non-overlapping vertex sets KP_i such that (1) $V(G) = \bigcup_i KP_i$; and (2) for any k-clique in G, its vertex set is contained in exactly one KP_i for some i.*

By Definition 4, it is immediate that for each vertex $v \in KP_i$ for some i, $C_k(v, G) = C_k(v, G[KP_i])$, where $G[KP_i]$ is the subgraph induced by KP_i in G. Thus, finding the $\mathcal{D}_k(G)$ of G is equivalent to finding the $\mathcal{D}_k(G)$ on each k-clique-isolating partition of G. To obtain such a partition, a naive method is to initialize each vertex as an individual partition and run a k-clique listing algorithm and merge any two partitions that share one k-clique. Due to the large number of k-cliques, there will be even more "merge" operations, which incurs a large overhead. To address this, we propose KPComputation that exploits the SCT*-Index to efficiently compute the k-clique-isolating partition as outlined in Algorithm 3.

Algorithm 3: KPComputation

Input: G: the input graph; $SCT(G)$: the SCT-index of G
Output: $par[u]$: the partition ID for each $u \in V(G)$

1 $par[u] \leftarrow u$ for each $u \in V(G)$;
2 $depth[g] \leftarrow 0$ for each partition $g \in \{par[u] : u \in V(G)\}$;
3 **foreach** *root-to-leaf path* $\mathcal{P}$ **do**
4 $par \leftarrow \{$findPartition$(par[u]) : u \in V(\mathcal{P})\}$;
5 $g^* \leftarrow arg\ max_{g \in par} depth[g]$;
6 **foreach** $g \in par$ *and* $g \neq g^*$ **do**
7 $par[g] \leftarrow g^*$;
8 **if** $depth[g] == depth[g^*]$ **then**
9 increase $par[g^*]$ by 1;
10 **return** $par[u]$ for each $u \in V(G)$;
11 **Function** findPartition(u):
12 **if** $par[u] == u$ **then**
13 $par[u] =$ findPartition$(par[u])$;
14 **return** $par[u]$;

Algorithm 3 adopts the disjoint set data structure [42] to dynamically maintain the k-clique-connectivity. For each vertex u, $par[u]$ stands for the parent node of u in the disjoints set forest and the findPartition procedure returns the root node of the tree where u resides. Algorithm 3 exploits an important structural characteristic of the SCT*-Index: each root-to-leaf path is a compressed representation of k-cliques that reside in the same k-clique isolating partition because they share the same hold vertices. Thus, instead of visiting all k-cliques and merging the partitions sharing the same k-cliques, KPComputation visits all root-to-leaf paths and merges the partitions containing the vertices on the same path. Note that union-by-rank (Lines 6-9) and path-compression (Line 13) optimizations are implemented. With these optimizations, findPartition takes near-constant time.

When the k-clique-isolating partition is obtained, we can obtain an upper bound on the maximum k-clique density for each partition via the following lemma.

Lemma 3. *Given a subgraph $G' \subseteq G$, the k-clique density $\rho_k(G')$ of G' must not exceed $\bar{\rho} = max_{v \in G} |C_k(v, G)|/k$.*

Proof. By Definition 1, $\rho_k(G') = \frac{|C_k(G')|}{|V(G')|}$. Since each k-clique has k vertices, $|C_k(G')| = \sum_{v \in V(G')} |C_k(v, G')|/k$. In addition, $|C_k(v, G')| \leq |C_k(v, G)| \leq max_{v \in G} |C_k(v, G)|$ since $G' \subseteq G$. Thus, $\rho_k(G') \leq \frac{max_{v \in G} |C_k(v,G|) \times |V(G')|}{k \times |V(G')|} = max_{v \in G} |C_k(v, G)|/k$. □

The above lemma allows us to use any provisionally obtained k-clique-density ρ' to prune some k-clique-isolating partitions. Specifically, if there exists a partition KP_j with $max_{v \in KP_j} |C_k(v, G)|/k < \rho'$, then KP_j can be safely removed because any subgraph of KP_j cannot have a k-clique density better than ρ'.

Clique-engagement-based reduction. In this part, we discuss how the k-clique engagement of a vertex, i.e., the number of k-cliques containing a vertex, can be used for graph reduction. Intuitively, when $|C_k(v, G)|$ is large, the vertex v is more likely to be included in the optimal solution and vice versa. Previous work [22] also points out this observation and proposes the (k', Ψ)-core model to control the k-clique engagement of the vertices. As reviewed in Section 3, one important finding is that the optimal solution must reside in a (k', Ψ)-core for some k' (Lemma 1). However, computing the (k', ψ)-core for graph reduction involves visiting all k-cliques for many iterations. This incurs huge overhead especially when the k-cliques are computed from scratch repeatedly as in [22]. Motivated by this, we derive the following lemma.

Lemma 4. *Given any subgraph $G' \subseteq G$ such that $\rho_k(G') = \rho'$, $\mathcal{D}_k(G)$ is contained in the subgraph $G_{\rho'}$, which is induced by the vertices with its k-clique engagement $\geq \lceil \rho' \rceil$. Here $G_{\rho'}$ is referred to the search scope w.r.t. the density ρ'.*

Proof. By Lemma 1, the k-clique densest subgraph resides in the (k', Ψ)-core where $k' = \lceil \rho_{opt} \rceil$. By the nested property of (k', Ψ)-core, (k', Ψ)-core is contained in (k'', Ψ)-core, where $k'' = \lceil \rho' \rceil \leq k'$. By Definition 3, (k'', Ψ)-core is contained by $G_{\rho'}$, which completes the proof. □

Based on the above lemma, as SCTL runs, the sub-optimal density ρ' increases and the resulting $G_{\rho'}$ shrinks. In the meantime, the k-clique engagement of the vertices in $G_{\rho'}$ can be easily updated by only visiting the k-cliques in $G_{\rho'}$. Specifically, we can avoid visiting any root-to-leaf path containing any hold vertex outside of $G_{\rho'}$ (i.e., any hold vertex v s.t. $|C_k(v, G)| < \lceil \rho' \rceil$). For each visited root-to-leaf path $\mathcal{P}$, we remove the pivot vertices that do not reside in $G_{\rho'}$. For each remaining hold vertex $v \in V_h(\mathcal{P})$, we increase $|C_k(v, G_{\rho'})|$ by $\binom{|V_p(\mathcal{P})|}{k-|V_h(\mathcal{P})|}$, since each k-clique contains all hold vertices and $k - |V_h(\mathcal{P})|$ pivot vertices on the root-to-leaf path. For each remaining pivot vertex $v \in V_p(\mathcal{P})$, we increase $|C_k(v, G_{\rho'})|$ by $\binom{|V_p(\mathcal{P})|-1}{k-|V_h(\mathcal{P})|-1}$ for the similar reason.

Example 4. *On the graph in Figure 1, we illustrate how the proposed graph reduction optimizations shrink the search scope for $\mathcal{D}_k(G)$. Initially, the k-clique-isolating partitions can be computed by calling* KPComputation, *which divides the graph into KP_1 and KP_2. In KP_1, the maximum per-vertex-k-clique count is $|C_k(v_2, G)| = 9$. In KP_2, the maximum per-vertex-k-clique count is $|C_k(v_8, G)| = 6$. By Lemma 3, the k-clique density upper bounds of KP_1 and KP_2 are computed to be $9/3 = 3$ and $6/3 = 2$, respectively. Within a few iterations,* SCTL *can quickly find a sub-optimal solution induced by the vertices $\{v_2$-$v_{12}\}$, which has a k-clique density of $\frac{23}{11}$. Based on clique-connectivity-based*

graph reduction, the KP_2 partition can be pruned since its density upper bound is smaller than the sub-optimal density. In addition, vertex v_1 can also be discarded since $|C_k(v_1, G)| = 1 < \lceil \frac{23}{11} \rceil$ by Lemma 4.

5.2 Batch processing k-cliques

Although graph reduction techniques can significantly reduce the search scope for $\mathcal{D}_k(G)$, each k-clique in G still needs to be visited once in each iteration (Algorithm 2 Lines 5-7). Since a root-to-leaf path $\mathcal{P}$ organizes multiple k-cliques, a natural question arises: "can we distribute the weight increase (i.e., the number of k-cliques in $\mathcal{P}$) to the vertices in $\mathcal{P}$ in a batch manner"? Consider the following extreme example on a path $\mathcal{P}$. If a vertex $u \in \mathcal{P}$ has weight zero while other vertices have very large weights, it is likely that u is always the vertex with the minimum weight when visiting all k-cliques under $\mathcal{P}$, and we can directly apply the total weight increase to u. However, in more complicated cases, we need to consider that the vertex with the minimum weight can be changed dynamically. To address this issue, we propose the `BatchUpdate` algorithm (as shown in Algorithm 4) that identifies the fundamental cases where batch updating vertex weights is possible and recursively breaks down the original problem into these cases.

Algorithm 4: `BatchUpdate`

Input: $\mathcal{P}$: a root-to-leaf path; lim: number of remaining k-cliques
Output: r: the vertex weights

1 **if** $lim \leq 0$ **then**
2 return;
3 Compute min_h, min_p, v^h_{min}, v^p_{min}, ck_{min} and gap by scanning $V_h(\mathcal{P})$ and $V_p(\mathcal{P})$;
4 **if** $lim == |C_k(\mathcal{P})|$ **then**
5 **if** $min_h < min_p$ **then**
6 $r(v^h_{min}) \leftarrow r(v^h_{min}) + min(ck_{min}, gap)$;
7 `BatchUpdate` $(\mathcal{P}, |C_k(\mathcal{P})| - min(ck_{min}, gap))$;
8 **else**
9 $r(v^p_{min}) \leftarrow r(v^p_{min}) + min(ck_{min}, gap)$;
10 **if** $ck_{min} > gap$ **then**
11 move v^p_{min} from $V_p(\mathcal{P})$ to $V_h(\mathcal{P})$ and call `BatchUpdate` $(\mathcal{P}, ck_{min} - gap)$;
12 **if** $|V(\mathcal{P})| - 1 \geq k$ **then**
13 `BatchUpdate` $(\mathcal{P} \setminus \{v^p_{min}\}, |C_k(\mathcal{P})| - ck_{min})$;
14 **if** $lim < |C_k(\mathcal{P})|$ **then**
15 $min^* = min(ck_{min}, gap, lim)$;
16 **if** $min_h < min_p$ **then**
17 $r(v^h_{min}) \leftarrow r(v^h_{min}) + min^*$;
18 `BatchUpdate` $(\mathcal{P}, lim - min^*)$;
19 **else**
20 $r(v^p_{min}) \leftarrow r(v^p_{min}) + min^*$;
21 **if** $gap == min^*$ **then**
22 move v^p_{min} from $V_p(\mathcal{P})$ to $V_h(\mathcal{P})$ and call `BatchUpdate` $(\mathcal{P}, min(lim, ck_{min}) - gap)$;
23 remove v^p_{min} in $V_h(\mathcal{P})$;
24 **if** $|V(\mathcal{P})| - 1 \geq k$ **then**
25 `BatchUpdate` $(\mathcal{P} \setminus \{v^p_{min}\}, lim - ck_{min})$;
26 **return** $r(u)$ for each $u \in V(\mathcal{P})$;

Given a root-to-leaf path $\mathcal{P}$, Algorithm 4 handles the k-cliques in $\mathcal{P}$ in batch and makes recursive calls on the number of k-cliques that remain to be processed in $\mathcal{P}$ (denoted by lim). Initially, lim is set to $|C_k(\mathcal{P})|$, where $C_k(\mathcal{P})$ represents the set of k-cliques in $\mathcal{P}$. Also, Algorithm 4 scans the vertices in $\mathcal{P}$ (Line 3) to compute:
(1) $\underline{min_h}$ and $\underline{second_h}$: the top-2 minimum vertex weights in $V_h(\mathcal{P})$; $\underline{v^h_{min}}$: the vertex with the minimum weight in $V_h(\mathcal{P})$;
(2) $\underline{min_p}$ and $\underline{second_p}$: the top-2 minimum vertex weights in $V_p(\mathcal{P})$; $\underline{v^p_{min}}$: the vertex with the minimum weight in $V_p(\mathcal{P})$;
(3) $\underline{w_{min}}$ and $\underline{w_{second}}$: the top-2 minimum vertex weights in $V(\mathcal{P})$.

For ease of presentation, we organize Algorithm 4 in the case when the vertex with the minimum weight is unique and discuss the general cases in the end. After identifying the vertex with the minimum weight, Algorithm 4 always processes the k-cliques containing the vertex first. We use gap to represent the difference between the top-2 minimum vertex weights in $V(\mathcal{P})$ (i.e., $gap = |w_{second} - w_{min}|$). In addition, we use ck_{min} to represent the number of k-cliques containing the vertex with the minimum weight:

$$ck_{min} = \begin{cases} \binom{|V_p(\mathcal{P})|}{k-|V_h(\mathcal{P})|} & min_h < min_p \\ \binom{|V_p(\mathcal{P})|-1}{k-|V_h(\mathcal{P})|-1} & otherwise \end{cases}$$

Since gap and lim restrict the largest weight increase that can be applied to the vertex with the minimum weight, the weight increase actually applied is the minimum of lim, ck_{min} and gap. Based on whether $lim = |C_k(\mathcal{P})|$ and the type of the vertex with the minimum weight, Algorithm 4 includes four cases as follows. In Cases 1-2, the vertex with the minimum weight is a hold vertex and in Cases 3-4 it is a pivot vertex.

Case 1: $lim = |C_k(\mathcal{P})|$ and $min_h < min_p$ (Lines 5-7). Since $lim = |C_k(\mathcal{P})| \geq ck_{min}$, $min(ck_{min}, gap)$ is the weight increase that is applied to v^h_{min} (Line 6). The remaining $|C_k(\mathcal{P})| - min(ck_{min}, gap)$ k-cliques are handled by a recursive call (Line 7).

Case 2: $lim < |C_k(\mathcal{P})|$ and $min_h < min_p$ (Lines 15-18). $lim < |C_k(\mathcal{P})|$ indicates that `BatchUpdate` is being recursively called by itself. Since min^* is the smallest among ck_{min}, gap, and lim, it is the weight increase applied to v^h_{min} (Line 17). The remaining $lim - min^*$ k-cliques are handled by a recursive call (Line 18).

Case 3: $lim = |C_k(\mathcal{P})|$ and $min_h \geq min_p$ (Lines 9-13). In this case, we focus on processing the k-cliques containing v^p_{min} first. Since $lim = |C_k(\mathcal{P})| \geq ck_{min}$, we apply the weight increase of $min(ck_{min}, gap)$ to v^p_{min} (Line 9). Note that when $ck_{min} > gap$ (Lines 10-11), only gap k-cliques are handled and the remaining k-cliques containing v^p_{min} still need to be processed. By moving v^p_{min} into $V_h(\mathcal{P})$, the recursive call will only handle the remaining $ck_{min} - gap$ k-cliques containing v^p_{min} (Line 11). Lastly, we process the k-cliques not containing v^p_{min} via a recursive call (Lines 12-13).

Case 4: $lim < |C_k(\mathcal{P})|$ and $min_h \geq min_p$ (Lines 20-25). This case is similar to Case 3. The difference is the number of k-cliques processed is capped at lim.

In each of these cases, the weight increase of certain vertices can be pre-computed and applied immediately instead of being incremented by one repeatedly. During the execution of Algorithm 4, these cases transform from each other via recursive calls, which significantly reduces the total number of weight updates in SCTL. Note that these cases handle the scenario when the vertex with the

minimum weight is unique. When the vertices with the minimum weight only contain $|S^h_{min}|$ hold vertices, the total weight increase of these vertices is $min(|S^h_{min}|\times|w_{min}-w_{second}|, lim, |C_k(\mathcal{P})|)$ and we apply such a change evenly to these hold vertices. When the vertices with the minimum weight contain pivot vertices, we process these pivot vertices one by one. Since in the worst case, BatchUpdate needs as many weight updates as the number of k-cliques under the root-to-leaf path $\mathcal{P}$, its time complexity is $O(|C_k(\mathcal{P})|)$.

Example 5. *We show how Algorithm 4 updates the vertex weights when processing the root-to-leaf path $\mathcal{P}$ = < $root, v_{12}, v_{11}, v_8, v_9, v_{10}$ > of the SCT*-Index in Figure 2 ($k = 3$). After the first iteration, the vertex weights in the path are: $r(v_{12}) = 1$, $r(v_{11}) = 4$, $r(v_8) = 4$, $r(v_9) = 4$ and $r(v_{10}) = 5$. In the following, we show the vertex weight updates in the second iteration. Initially, lim is set to $|C_k(\mathcal{P})| = \binom{4}{3-1} = 6$. Since the hold vertex v_{12} has the minimum weight, Algorithm 4 enters Case 1 and update $r(v_{12})$ to 4, which is the second minimum weight. Then, Algorithm 4 processes the remaining k-cliques via a recursive call with $lim = 3$. In the recursive call, Algorithm 4 enters Case 4 since there exists a pivot vertex v_9 with the minimum weight 4. In this case, a recursive call is invoked to handle the k-cliques containing v_9, and another recursive call is used to handle the other k-cliques not containing v_9. The first recursive call results in two more recursive calls of Case 4. Note that these three recursive calls increase the weights of v_8, v_9, and v_{11} by one, respectively. Then, Algorithm 4 terminates. In total, the resulting total number of vertex weight updates is 4, reduced from the number of k-cliques of 6.*

5.3 The SCTL* algorithm

Algorithm 5: SCTL*

Input: G: the input graph; T: the number of iterations, $SCT(G)$: the SCT-index of G
Output: $\mathcal{G}$: an approximate solution on G

1 $\rho' \leftarrow \frac{\binom{M}{k}}{M}$, $r(u) \leftarrow 0$ for all $u \in V(G)$;
2 Call KPComputation to compute $par[v]$ for all $v \in V(G)$;
3 **foreach** $t \leftarrow 1, 2, 3, ...T$ **do**
4 **foreach** *valid root-to-leaf path* $P \in SCT(G)$ **do**
5 **if** *there exists* $v \in V(\mathcal{P})$ *s.t.* $\bar{\rho}(par[v]) \leq \rho'$ **then**
6 continue;
7 $V_h \leftarrow$ the hold vertices in P with $|Ck(v, G_{\rho'})| \geq \lceil \rho' \rceil$;
8 $V_p \leftarrow$ the pivot vertices in P with $|Ck(v, G_{\rho'})| \geq \lceil \rho' \rceil$;
9 increase $|Ck(v, G_{\rho'})|$ by $\binom{|V_p|}{k-|V_h|}$ for each $v \in V_h$;
10 increase $|Ck(v, G_{\rho'})|$ by $\binom{|V_p|-1}{k-|V_h|-1}$ for each $v \in V_p$;
11 BatchUpdate $(\mathcal{P}, |C_k(\mathcal{P})|)$;
12 $\rho' \leftarrow$ the k-clique density of the current solution;
13 run Lines 6-10 of Algorithm 1;
14 **return** $\mathcal{G}$;

We apply the above optimizations to SCTL and propose the SCTL* algorithm as shown in Algorithm 5. Note that we use ρ' to denote the best currently found k-clique density. ρ' is initialized to $\frac{\binom{M}{k}}{M}$, where M is the maximum clique size fetched from the SCT*-Index. The vertex weights are initialized to zero (Line 1). Then, the KPComputation algorithm is called to compute the k-clique-isolating partitions in G and $par[u]$ represents the partition ID of a vertex u (Line 2). Just like SCTL, SCTL* refines the vertex weights as it visits the k-cliques for T iterations (Lines 3-12). The difference is that, due to clique-connectivity and clique-engagement-based graph reduction, SCTL* only needs to process the k-cliques on progressively smaller search scope $G_{\rho'}$, where $G_{\rho'}$ is the search scope for $\mathcal{D}_k(G)$ resulting from the sub-optimal density ρ'. Specifically, if any vertex u in root-to-leaf path $\mathcal{P}$ satisfies $\bar{\rho}(par[v]) \leq \rho'$, then the whole path is discarded because this path resides in a k-clique-isolating-partition whose density upper bound is dominated by ρ' (Lines 5-6). Moreover, SCTL* only considers the k-cliques formed by the hold vertices and pivot vertices with $|Ck(v, G_{\rho'})| \geq \lceil \rho' \rceil$ (Lines 7-8). In addition, to process these significantly reduced k-cliques, SCTL* calls BatchUpdate algorithm to handle the k-cliques under the same root-to-leaf path together, which further reduces the number of vertex weight updates. Note that the k-clique engagement of the vertices in $G_{\rho'}$ are updated when visiting each root-to-leaf path without increasing the time complexity (Line 9-10).

Complexity analysis. The dominating cost of SCTL* is still incurred by SCT*-Index traversal and vertex weight updates. Our SCT*-Index-based graph reduction reduces the number of k-cliques processed from $|C_k(G)|$ to $|C_k(G_t)|$, where G_t is the search scope for $\mathcal{D}_k(G)$ in iteration t. Note that by utilizing BatchUpdate, the number of vertex weight updates is much smaller than $|C_k(G_t)|$. Like SCTL, SCTL* needs an additional scan of k-cliques to recover the k-clique density of the approximate solution. The total time complexity is $O((T+1)\times\sum_{\mathcal{P}}|V(\mathcal{P})| + \sum_{t=1}^{T}|C_k(G_t)|) + |C_k(G_T)|$.

6 SAMPLING-BASED SOLUTIONS

In the literature, the sampling strategy has been applied to get approximate solutions of the k-clique densest subgraph more quickly [37, 41]. However, existing sampling-based approximate algorithms rely on enumerating the k-cliques for sampling and recovering the k-clique density, which is infeasible for large-scale graphs. In this part, we propose the SCTL*-Sample algorithm, which exploits the structure of SCT*-Index to avoid enumerating all k-cliques for improved scalability. In addition, we propose an exact algorithm SCTL*-Exact that utilizes a near-optimal solution obtained by SCTL*-Sample to reduce the search scope for $\mathcal{D}_k(G)$.

6.1 The sampling-based approximate algorithm

Based on the SCT*-Index, the advanced sampling-based algorithm SCTL*-Sample involves the following two optimizations. First, we can pre-compute how many k-cliques we need from each root-to-leaf path and only visit the k-cliques that will be sampled. Second, we can easily compute the k-clique density of the subgraph induced by some vertex set via the SCT*-Index. In either situation, we can avoid listing all k-cliques in G. The SCTL*-Sample algorithm consists of three stages: (1) the sampling stage; (2) the weight refinement stage; and (3) the recovery stage. The details of the algorithm are summarized in Algorithm 6.

In the sampling stage (Lines 1-3), we obtain a sample of k-cliques of size σ from the SCT*-Index. To ensure the sampled k-cliques are distributed evenly under each root-to-leaf path, we pre-compute a sampling ratio, which is the ratio of k-cliques we need from

Algorithm 6: SCTL*-Sample

Input: G: the input graph; T: the number of iterations, $SCT(G)$: the SCT-index of G, σ: the number of k-cliques sampled
Output: $\mathcal{G}$: an approximate solution on G

1 $\mathcal{K} \leftarrow \emptyset$;
2 **foreach** *valid root-to-leaf path* $\mathcal{P}$ **do**
3 add $|C_k(\mathcal{P})| \frac{\sigma}{|C_k(G)|}$ k-cliques from $C_k(\mathcal{P})$ to $\mathcal{K}$;
4 $\rho' \leftarrow 0$, $r(u) \leftarrow 0$ for each $u \in V(\tilde{G})$;
5 **foreach** $t \leftarrow 1, 2, 3, \ldots T$ **do**
6 **foreach** *sampled* k-*clique* $C \in \mathcal{K}$ **do**
7 **if** $|Ck(v, \tilde{G})| \geq \lceil \rho' \rceil$ *for each* $v \in C$ **then**
8 $u^* = arg_{u \in V(C)} min(r(u))$;
9 increase $r(u^*)$ by 1;
10 update $|Ck(v, \tilde{G})|$ for each $v \in C$;
11 $\rho' \leftarrow$ the k-clique density of the approximate solution in $\tilde{G}$;
12 $\tilde{\mathcal{G}} \leftarrow$ the approximate solution on the sampled subgraph $\mathcal{G}$;
13 use SCT*-Index to recover the k-clique density of $G[V(\tilde{\mathcal{G}})]$;
14 $\mathcal{G} \leftarrow G[V(\tilde{\mathcal{G}})]$;
15 **return** $\mathcal{G}$;

$C_k(G)$, i.e. $\frac{\sigma}{|C_k(G)|}$. Then, under each processed root-to-leaf path, we aim to sample $|C_k(\mathcal{P})| \frac{\sigma}{|C_k(G)|}$ k-cliques and store them in main memory (Line 3). Note that $|C_k(\mathcal{P})|$ is computed from the formula in Lemma 2. We terminate this process when the sample size reaches σ. With the SCT*-Index, we are able to obtain a sample of k-cliques efficiently without visiting all k-cliques.

In the weight refinement stage (Lines 4-11), we scan the sampled k-cliques for T iterations and apply the same vertex weight update strategy to the sampled k-cliques as in SCTL*. We use $\tilde{G}$ to denote the subgraph induced by the vertices in the sampled k-cliques. To apply clique-engagement-based graph reduction in Section 5 on $\tilde{G}$, we compute the k-clique-engagement of the vertices in $\tilde{G}$ (i.e., $|Ck(v, \tilde{G})|$) during sampling. We use ρ' to store the best currently obtained k-clique density (Line 11). By Lemma 4, we only visit the k-cliques whose vertices satisfy $|Ck(v, \tilde{G})| \geq \lceil \rho' \rceil$ (Line 7). In this way, we need not visit all sampled k-cliques in each iteration, which improves efficiency.

In the recovery stage (Lines 12-14), we first compute the approximate solution on the sampled subgraph $\tilde{G}$ (denoted by $\tilde{\mathcal{G}}$) and then recover the k-clique density of the subgraph induced by the same vertices on G (denoted by $G[V(\tilde{\mathcal{G}})]$) using the SCT*-Index (Lines 13-14). In Line 12, we first sort the vertices in non-increasing order of their weights ($r(u)$). For each $i \leq n$, we compute y_i, which is the number of k-cliques contained in the subgraph induced by the first i vertices in the order, by scanning all sampled k-cliques. Then, we find s such that $\frac{\sum_{i \leq s} y_i}{s}$ is maximized and return the subgraph induced by the first s vertices as $\tilde{\mathcal{G}}$. In Line 13, we use the SCT*-Index to count the number of k-cliques in $G[V(\tilde{\mathcal{G}})]$ (i.e., $|C_k(G[V(\tilde{\mathcal{G}})])|$). For each root-to-leaf path $\mathcal{P}$, we remove the vertices not in $G[V(\tilde{\mathcal{G}})]$ and increment $|C_k(G[V(\tilde{\mathcal{G}})])|$ by $\binom{p'}{k-h'}$ by Lemme 2, where p' and h' are the number of remaining pivot and hold vertices in $\mathcal{P}$. The k-clique density of $G[V(\tilde{\mathcal{G}})]$ is computed as $\frac{|C_k(G[V(\tilde{\mathcal{G}})])|}{|V(\tilde{\mathcal{G}})|}$. In this way, we exploit the structural characteristics of the SCT*-Index and avoid visiting all k-cliques to recover the k-clique density in the original graph.

Utilizing the SCT*-k'-Index. Note that on some large-scale graphs, only the SCT*-k'-Index can be built for some k'. In this case, we can still run SCTL*-Sample using the root-to-leaf paths in SCT*-k'-Index and obtain reasonable approximations for arbitrary k. This is because most k-cliques in $\mathcal{D}_k(G)$ generally come from larger cliques, whose enumeration is likely to be supported by SCT*-k'-Index.

Complexity analysis. In the sampling stage, Algorithm 6 visits all valid root-to-leaf paths in SCT*-Index that contain k-cliques and store σ of them. Traversing the SCT*-Index takes $O(\sum_{\mathcal{P}} |V(\mathcal{P})|)$ time and storing σ k-cliques takes $O(k\sigma)$ time. The time complexity of the weight refinement stage is $O(Tk\sigma)$, in which the sampled k-cliques are scanned for T rounds. In the recovery stage, Algorithm 6 visits all sampled k-cliques again to obtain the approximate solution on $\tilde{G}$. Then, it computes the number of k-cliques of the approximate solution on G by selectively visiting some root-to-leaf paths, which takes $O(\sum_{\mathcal{P}} |V(\mathcal{P})|)$ time. Overall, the time complexity of SCTL*-Sample is $O(\sum_{\mathcal{P}} |V(\mathcal{P})| + (T+2)k\sigma)$. The dominating space cost is incurred by the SCT*-Index.

Remark 1. SCTL*-Sample *is an approximate algorithm like* SCTL* *with improved scalability via sampling. The difference is that when an approximate solution is obtained from* SCTL*, *an upper bound on the maximum k-clique density can be derived, which results in a lower bound for the approximation ratio. For* SCTL*-Sample, *the upper bound derived is actually with high probability instead of deterministic [17, 41].*

6.2 The sampling-based exact algorithm

Algorithm 7: SCTL*-Exact

Input: G: the input graph; $SCT(G)$: the SCT-index of G, σ: the number of k-cliques sampled
Output: $\mathcal{D}_k(G)$: a k-clique densest subgraph of G

1 run SCTL*-Sample to obtain ρ' ;
2 $G_{\rho'} \leftarrow$ the subgraph induced by the vertices $C_k(v, G) \geq \lceil \rho' \rceil$;
3 **while** $G_{\rho'}$ *is reducing* **do**
4 update $C_k(v, G_{\rho'})$ for all $v \in V(G_{\rho'})$;
5 $T \leftarrow 10$;
6 **while** $\mathcal{D}_k(G)$ *is not found* **do**
7 run SCTL* for T iterations;
8 **if** *The approximate solution is optimal* **then**
9 break;
10 Else $T \leftarrow 2T$;
11 **return** $\mathcal{D}_k(G)$;

In this part, we introduce the SCT*-Index-based exact algorithm SCTL*-Exact, which utilizes SCTL*-Sample to quickly obtain a near-optimal solution and drastically reduces the search scope.

As shown in Algorithm 7, SCTL*-Exact calls SCTL*-Sample to obtain a near-optimal k-clique density ρ' quickly (Line 1). Since ρ' is very close to the maximum k-clique density, we apply clique-engagement-based graph reduction with ρ' and reduce the search scope $G_{\rho'}$ until it stops shrinking (Lines 2-4). In this way, a very tight search scope for $\mathcal{D}_k(G)$ is obtained and from this point on

only the k-cliques in $G_{\rho'}$ need to be considered. Then, we run SCTL* for an increasing number of iterations (Lines 5-10). We utilizes the *improved Goldberg's condition* from KCL-Exact in [41] and max-flow to test if the current approximate solution is optimal. Since only the k-cliques in $G_{\rho'}$ are considered, running max-flow incurs much less memory compared to previous exact algorithms like CoreExact and KCL-Exact.

7 EXPERIMENTAL EVALUATIONS

In this section, we conduct experiments to evaluate the effectiveness and efficiency of our algorithms versus the state-of-the-arts.

7.1 Experimental settings

Algorithms. We extensively evaluate the following algorithms: *Approximate algorithms:* 1) CoreApp: the (k, ϕ)-core-based algorithm in [22]; 2) KCL: the convex-programming-based algorithm in [41]; 3) KCL-Sample: KCL with the sampling strategy in [41]; 4) SCTL: our basic SCT*-Index-based algorithm in Section 4.2; 5) SCTL$^+$: SCTL with graph reduction optimizations; 6) SCTL*: SCTL with all optimizations proposed in Section 5.3; 6) SCTL*-Sample: SCTL* with the sampling strategy proposed in Section 6.1.
Exact algorithms: 1) The exact algorithm KCL-Exact in [41]. 2) Our SCT*-Index-based exact algorithm proposed in Section 6.2.

We also report the construction time and size of the SCT*-Index that is used for our SCT-index-based algorithms. All algorithms are implemented in C++ except for CoreApp, for which we directly use its original Java implementation from [22]. The experiments are run on a Linux server with 2 × Intel Xeon E5-2698 processor (2.20GHz, 40 Cores) and 512GB main memory. *We terminate an algorithm if the running time is more than* 10^5 *seconds by default.*
Datasets. In our experiments, we use 12 real-world datasets including all the datasets used in [41]. All the datasets used here can be found in SNAP (http://snap.stanford.edu/data/). The summary of datasets is shown in Table 2. $|V(G)|$ denotes the number of vertices, and $|E(G)|$ represents the number of edges. The types of vertices and edges are also shown in the table. k_{max} represents the size of the maximum k-clique.

Table 2: Summary of datasets

Dataset	$\|E\|$	Type of E	$\|V\|$	Type of V	k_{max}
Email	183,831	Communication	36,692	Email	20
Amazon	925,872	Purchasing	334,863	Item	7
loc-gowalla	950,327	Share	196,591	User	29
DBLP	1,049,866	Authorship	425,957	Papers	114
road-CA	2,766,607	Connection	1,965,206	Endpoints	4
WikiTalk	4,659,565	Edit	2,394,385	Users	26
Youtube	2,987,624	Friendship	1,134,890	Users	17
as-skitter	11,095,298	Connection	1,696,415	IP addresses	67
soc-pokec	22,301,964	Friendship	1,632,803	Users	29
LiveJournal	34,681,189	Friendship	4,036,538	Users	327
Orkut	117,185,083	Friendship	3,072,627	Users	51
Friendster	1,806,067,135	Friendship	124,836,180	Users	129

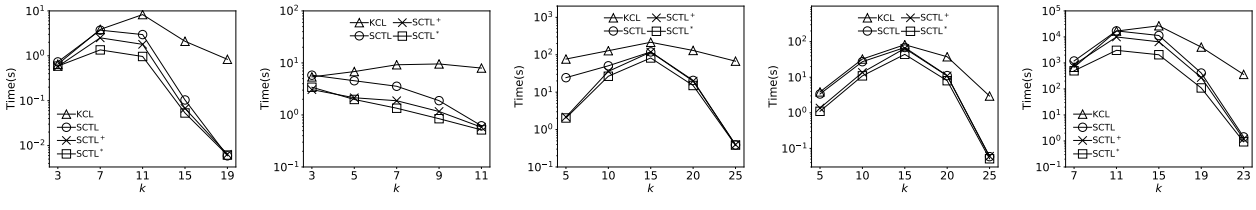
7.2 Performance of algorithms

Evaluate the effectiveness and efficiency of SCTL*. As shown in Table 3, we report the representative query results ($k = 15$) of the three approximation algorithms CoreApp, KCL, and SCTL* and their total query time for all values of k on 5 datasets. Note that we run KCL and SCTL* for 10 iterations. On datasets where exact solutions are available (i.e., Email, loc-gowalla, and Youtube), we report the actual approximation ratios. Otherwise, we use the upper bounds obtained from our SCTL* algorithm to estimate the approximation ratios as on WikiTalk and soc-pokec. We observe that KCL and SCTL* are able to achieve near-optimal approximations while CoreApp yields worse approximation ratios. This is because KCL and SCTL* are guaranteed to converge to the optimal solution with a sufficient number of iterations but CoreApp always returns the (k'_{max}, Ψ)-core with a theoretical approximation ratio of $1/k$. Also, CoreApp incurs high computation cost since it needs to repeatedly call the KClist algorithm to compute the (k'_{max}, Ψ)-core. We also report the construction time of the SCT*-Index and the ratio of the number of tree nodes in the SCT*-Index to the number of edges in the graph. We can see that the SCT*-Index can be built within reasonable time and space for these datasets. With such an index, we observe that SCTL* is significantly faster than KCL and CoreApp because (1) it directly "reads off" the k-cliques from the SCT*-Index instead of computing from scratch; and (2) its efficiency is further improved by graph reduction and batch processing optimizations. In the remaining experiments, we focus on evaluating the convex-programming-based solutions (i.e., KCL and the SCT*-Index-based algorithms) since the efficiency and effectiveness of these algorithms are much better than CoreApp.
Evaluate the effect of k. In Figure 4, we report the running time of KCL, SCTL, SCTL$^+$, and SCTL* for different values of k on 5 datasets. Overall, the SCT*-Index-based algorithms significantly outperform KCL with different k values, and the proposed graph reduction and batch processing optimizations incrementally speed up the SCTL algorithm as expected. Specifically, when k approaches k_{max}, the SCT*-Index-based algorithms are faster than KCL by up to two orders of magnitude. For example, SCTL* achieves a speedup of 733× and 135× on Wikitalk ($k = 19$) and Email ($k = 23$), respectively. This is because the KClist algorithm adopted by KCL is originally designed for sparse graphs with small k values (e.g., $k < k_{max}/2$), while SCTL and SCTL* can fetch the few k-cliques very quickly from the SCT*-Index when k is large. In addition, the proposed optimizations for SCTL are usually more effective when k approaches $\frac{k_{max}}{2}$, which is generally when the number of k-cliques is very large.

As shown in Figure 5, we also report the k-clique densities of the approximate solutions from KCL, SCTL, and SCTL* after 10 iterations as k varies on Email and Youtube. As expected, KCL, SCTL, and SCTL* can produce near-optimal approximations and the proposed optimizations do not affect the effectiveness of SCTL.
Evaluate the effect of graph reduction and batch processing. In Table 4, we report some statistics to illustrate the effectiveness of the proposed optimizations in SCTL*. On Email and Youtube, we choose two representative k values and run SCTL* for 10 iterations. G_T represents the search scope for the k-clique densest subgraph before entering iteration T. $\frac{|C_k(G_T)|}{|C_k(G)|}$ denotes the ratio of the number of k-cliques contained by G_T and those in G. $\frac{\#\text{updates}}{|C_k(G)|}$ is the ratio of the number of actual vertex weight updates by SCTL* in iteration T and the number of k-cliques. We can observe that as T increases, G_T is reduced progressively, reflected by its decreasing number of

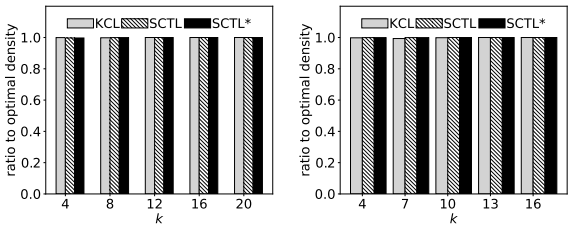
Table 3: Index build time and query time of approximation algorithms ($T = 10$)

Dataset	SCT*-Index		Query time (s) and approximate ratios when $k = 15$			Total query time (s) for all k		
	Time (s)	#tree nodes / m	CoreApp	KCL	SCTL*	CoreApp	KCL	SCTL*
Email	0.40	21.67	196.74 (0.82)	2.29 (1.00)	**0.05 (1.00)**	2706.91	64.58	**11.73**
loc-gowalla	2.08	15.58	1402.836 (0.70)	81.87 (1.00)	**43.67 (1.00)**	27677.80	809.27	**331.79**
WikiTalk	154.11	122.80	time out	20679.20 (≥ 0.96)	**1912.05** (≥ 0.96)	time out	time out	**22443.88**
Youtube	5.88	4.69	2973.58 (0.47)	5.47 (1.00)	**0.07 (1.00)**	time out	109.30	**15.71**
soc-pokec	48.62	27.38	13723.77 (≥ 0.97)	206.40 (≥ 0.97)	**82.22** (≥ 0.97)	time out	3164.05	**681.06**



(a) Email (b) Youtube (c) soc-Pokec (d) Gowalla (e) Wikitalk

Figure 4: Effect of k on the running time of KCL and SCT*-Index based approximate algorithms



(a) Email (b) Youtube

Figure 5: Effectiveness of KCL, SCTL, and SCTL*

Table 4: Effectiveness of the proposed optimizations

Dataset	k	$C_k(G)$	T	$\lvert V(G_T)\rvert$	$\lvert E(G_T)\rvert$	$\frac{\lvert C_k(G_T)\rvert}{\lvert C_k(G)\rvert}$	$\frac{\#\text{updates}}{\lvert C_k(G)\rvert}$
Email	7	1.63×10^7	1	903	60012	94.76%	14.91%
			6	837	18094	58.28%	9.02%
			10	293	15496	53.12%	7.44%
	11	8.86×10^6	1	383	22792	93.62%	31.77%
			6	328	6754	50.29%	18.67%
			10	135	6164	45.89%	18.78%
Youtube	7	7.96×10^6	1	1352	98812	92.48%	13.79%
			6	1175	15628	47.29%	6.17%
			10	251	13144	42.25%	5.96%
	11	7.17×10^5	1	260	14418	93.27%	35.57%
			6	236	4356	63.14%	24.95%
			10	102	3394	53.33%	23.71%

vertices and edges. The number of k-cliques enclosed by G_T also decreases steadily. These validate the effect of our SCT*-Index-based graph reduction optimization. Note that G_T is already significantly smaller compared to G before the first iteration because the maximum clique serves as a non-trivial approximate solution that can be used for graph reduction. In addition, we observe that the number of vertex weight updates by SCTL* is noticeably smaller than the number of k-cliques in G_T. This is because the batch processing optimization enables SCTL* to consider the k-cliques under the same root-to-leaf path together and avoid enumerating each of them.

Table 5: Comparison between KCL-Sample and SCTL*-Sample

Dataset	SCT*-Index		k	KCL-Sample		SCTL*-Sample	
	Time (s)	#tree nodes / m		Time (s)	Density	Time (s)	Density
Email	0.40	21.67	5	8.11	8.07×10^3	7.83	8.07×10^3
			10	21.87	5.75×10^4	17.72	5.75×10^4
			15	6.64	3.57×10^3	3.93	3.57×10^3
as-Skitter	122.31	86.67	5	86.14	1.12×10^6	19.22	1.12×10^6
			25	time out	–	26.26	2.36×10^{17}
			45	time out	–	32.95	3.13×10^{16}
			65	time out	–	2.81	1.33×10^2
DBLP	2.29	2.99	10	time out	–	20.44	5.97×10^{11}
			40	time out	–	36.28	8.27×10^{28}
			70	time out	–	47.26	7.00×10^{29}
			100	time out	–	57.69	2.74×10^{15}
Orkut	1207.05*	319.53	10	time out	–	101.22	4.10×10^{10}
			20	time out	–	83.49	1.25×10^{15}
			30	time out	–	75.58	1.42×10^{15}
			40	time out	–	52.66	1.14×10^{11}
Live-journal	3767.68*	294.62	50	time out	–	279.83	5.81×10^{58}
			100	time out	–	371.05	2.94×10^{86}
			150	time out	–	426.67	7.00×10^{97}
			200	time out	–	483.89	3.21×10^{95}
			250	time out	–	431.31	6.59×10^{78}
			300	time out	–	335.85	268×10^{42}
Friendester	5669.84*	2.59	25	time out	–	59.49	2.23×10^{24}
			50	time out	–	79.69	2.70×10^{29}
			75	time out	–	91.29	1.97×10^{29}
			100	time out	–	103.22	1.72×10^{27}
			125	time out	–	122.31	3.33×10^5

Evaluate the effect of sampling. We compare two sampling-based approximate algorithms (i.e., KCL-Sample and SCTL*-Sample) and report the running time and the resulting k-clique densities for varying values of k on 6 datasets in Table 5. For both algorithm, we always sample 10^7 k-cliques as done in [41]. We also report the SCT*-Index build time and the ratio of the number of tree

nodes in the SCT*-Index to the number of edges in the graph. Note that on graphs Orkut, Live-journal, and Friendester, we build SCT*-k'-Index and set k' set to 40, 326, and 128, respectively. Out of the 6 datasets with varying k values, the optimal solutions are only available on Email and DBLP, and our SCTL*-Sample is able to find these solutions quickly. This is because the k-clique densest subgraph on the sampled subgraph and that of the original graph usually highly overlap [37]. In addition, SCTL*-Sample is able to provide reasonable approximate solutions for all values of k on graphs with as many as 1 billion edges, while KCL-Sample is only feasible for limited k values or small graphs. This is because SCTL*-Sample avoids enumerating all k-cliques when extracting a sample of k-cliques and when recovering the approximate k-clique density on the original graph with the SCT*-Index, respectively. In addition, SCTL*-Sample applies the clique-engagement-based graph reduction to further reduce the number of sampled k-cliques visited in each iteration.

Table 6: Compare query time (s) of the exact algorithms

Dataset	k	KCL-Exact	SCTL*-Exact
Email	10	33.96	8.27
	15	4.43	4.37
	20	3.25	0.03
Youtube	10	32.18	5.97
	15	4.43	0.16
Orkut	48	time out	27.50
	49	time out	25.54
	50	time out	24.51
	51	time out	15.52
Live-Journal	327	out of memory	197.66

Compare to the state-of-the-art exact algorithms. We compare the running time of KCL-Exact and SCTL*-Exact on 4 datasets with some representative k values in Table 6. We can observe that SCTL*-Exact outperforms KCL-Exact in all settings. On small datasets like Email and Youtube, SCTL*-Exact achieves better efficiency because it uses sampling to obtain a near-optimal solution and a relatively small subgraph as the search scope for $\mathcal{D}_k(G)$. The dominating time cost for both algorithms is incurred when verifying that the current approximate solution is indeed optimal via a flow network. On Orkut, for the prescribed k values, KCL-Exact cannot finish within the time limit, while SCTL*-Exact can obtain the exact solutions efficiently. On Live-Journal, KCL-Exact runs out of memory from storing the k-cliques. We can observe that SCTL*-Exact achieves better efficiency while incurring less memory by focusing on the k-cliques in a reduced search scope.

8 RELATED WORK

Below we review the related works of the edge densest subgraph problem and the k-clique densest subgraph problem, which are closely related.

Edge densest subgraph. The edge densest subgraph problem (**EDS**) aims to find the subgraph with the maximum average degree [1, 3, 4, 7–9, 20, 30, 34, 39, 43]. An important finding in the **EDS** literature is that solving a parametric maximum flow problem can be used to verify if a subgraph is the densest subgraph [26]. This establishes a general framework of conducting a binary search on the maximum density and using a flow network as a verification tool for **EDS** and its variants [36, 45]. Generally, solving the maximum flow problem [14, 24, 27] can be time-consuming and such an exact algorithm becomes infeasible on large graphs. Approximation algorithms are usually proposed to improve efficiency [11, 17, 41, 46]. The peeling algorithm for k-core decomposition runs in linear time and provides a 2-approximation for the **EDS** problem [5, 11]. A recent work [46] studies the p-mean densest subgraph problem and proposes a generalized peeling algorithm with an approximation ratio of $1/(1+p)^{\frac{1}{p}}$, which reduces to **EDS** when p equals one. In addition, a recent line of research formulates the densest subgraph problem as a convex program and resorts to convex optimization algorithms to design approximation and exact algorithms [17, 35, 41].

k-clique densest subgraph. As a variant of **EDS**, the k-clique densest subgraph problem is proposed in an attempt to better detect "near-clique" subgraphs, which aims to maximize the average number of k-cliques per vertex over all subgraphs [45]. The classic framework of conducting a binary search for the maximum density via max-flow is extended to solve the k-clique densest subgraph problem [22, 37, 45], where a hyper-graph with the same vertices and the k-cliques as hyper-edges is considered. The exact algorithm under this framework has limited scalability due to the large number of k-cliques and the large size of the resulting flow network. To address this, a cohesive subgraph model (k', Ψ)-core is proposed in [22] to apply graph reduction, which allows the flow network to be built on progressively smaller subgraphs. The (k', Ψ)-core with the maximum k' itself also serves as a $\frac{1}{k}$ approximation. [37] adopts the same framework to compute both the k-clique densest subgraph and the (p, q)-biclique densest subgraph (on bipartite graphs). They discover that sampling a fraction of the k-cliques is enough to obtain a close approximate solution. In addition, convex programming based algorithms are proposed in [41], including a large memory Frank-Wolfe algorithm and a linear memory approximate algorithm KCL that converges to the optimal solution after scanning the k-cliques for sufficient iterations. Compared to existing solutions, our work can produce near-optimal solutions more efficiently via novel graph reduction and batch processing optimizations with the SCT*-Index. We also propose a sampling-based algorithm capable of providing approximate solutions for arbitrary k values on graphs at billion-scale, which also facilitates the search for the exact solution.

9 CONCLUSION

In this paper, we study the k-clique densest subgraph problem and provide more efficient and scalable algorithms. By adapting the succinct clique tree, we propose the SCT*-Index to store k-cliques. Based on the SCT*-Index, we propose the SCTL algorithm with near-optimal approximation ratios in practice, which is further sped up by novel graph reduction and batch processing techniques. We further push the efficiency boundary via sampling and our SCTL*-Sample algorithm can offer reasonable approximations for graphs at billion-scale, which also facilitates the search for the exact solution. Extensive experiments on 12 real-world graphs verify the efficiency and effectiveness of the proposed techniques.

REFERENCES

[1] Venkat Anantharam and Justin Salez. 2016. The densest subgraph problem in sparse random graphs. *The Annals of Applied Probability* 26, 1 (2016), 305–327.

[2] 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.

[3] Albert Angel, Nick Koudas, Nikos Sarkas, and Divesh Srivastava. 2012. Dense subgraph maintenance under streaming edge weight updates for real-time story identification. *arXiv preprint arXiv:1203.0060* (2012).

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

[5] Yuichi Asahiro, Kazuo Iwama, Hisao Tamaki, and Takeshi Tokuyama. 2000. Greedily finding a dense subgraph. *Journal of Algorithms* 34, 2 (2000), 203–221.

[6] Gary D Bader and Christopher WV Hogue. 2003. An automated method for finding molecular complexes in large protein interaction networks. *BMC bioinformatics* 4, 1 (2003), 1–27.

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

[8] 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.

[9] 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.

[10] Gregory Buehrer and Kumar Chellapilla. 2008. A scalable pattern mining approach to web graph compression with communities. In *Proceedings of the 2008 international conference on web search and data mining*. 95–106.

[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] Zi Chen, Long Yuan, Xuemin Lin, Lu Qin, and Jianye Yang. 2020. Efficient maximal balanced clique enumeration in signed networks. In *Proceedings of The Web Conference 2020*. 339–349.

[14] Boris V Cherkassky and Andrew V Goldberg. 1997. On implementing the push—relabel method for the maximum flow problem. *Algorithmica* 19, 4 (1997), 390–410.

[15] 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).

[16] 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.

[17] 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.

[18] Apurba Das, Seyed-Vahid Sanei-Mehri, and Srikanta Tirthapura. 2020. Shared-memory parallel maximal clique enumeration from static and dynamic graphs. *ACM Transactions on Parallel Computing (TOPC)* 7, 1 (2020), 1–28.

[19] 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.

[20] 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.

[21] David Eppstein and Darren Strash. 2011. Listing all maximal cliques in large sparse real-world graphs. In *International Symposium on Experimental Algorithms*. Springer, 364–375.

[22] Yixiang Fang, Kaiqiang Yu, Reynold Cheng, Laks V. S. Lakshmanan, and Xuemin Lin. 2019. Efficient Algorithms for Densest Subgraph Discovery. *Proc. VLDB Endow.* 12, 11 (2019), 1719–1732.

[23] 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.

[24] Giorgio Gallo, Michael D Grigoriadis, and Robert E Tarjan. 1989. A fast parametric maximum flow algorithm and applications. *SIAM J. Comput.* 18, 1 (1989), 30–55.

[25] 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*. Citeseer, 721–732.

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

[27] Andrew V Goldberg and Robert E Tarjan. 1988. A new approach to the maximum-flow problem. *Journal of the ACM (JACM)* 35, 4 (1988), 921–940.

[28] Bryan Hooi, Hyun Ah Song, Alex Beutel, Neil Shah, Kijung Shin, and Christos Faloutsos. 2016. Fraudar: Bounding graph fraud in the face of camouflage. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. 895–904.

[29] 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.

[30] Shuguang Hu, Xiaowei Wu, and TH Hubert Chan. 2017. Maintaining densest subsets efficiently in evolving hypergraphs. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. 929–938.

[31] Martin Jaggi. 2013. Revisiting Frank-Wolfe: Projection-free sparse convex optimization. In *International Conference on Machine Learning*. PMLR, 427–435.

[32] 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.

[33] 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.

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

[35] 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.

[36] 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.

[37] 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.

[38] Kevin A Naudé. 2016. Refined pivot selection for maximal clique enumeration in graphs. *Theoretical Computer Science* 613 (2016), 28–37.

[39] 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.

[40] Ahmet Erdem Sariyuce, C Seshadhri, Ali Pinar, and Umit V Catalyurek. 2015. Finding the hierarchy of dense subgraphs using nucleus decompositions. In *Proceedings of the 24th International Conference on World Wide Web*. 927–937.

[41] Bintao Sun, Maximilien Danisch, TH 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).

[42] Robert Endre Tarjan. 1975. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)* 22, 2 (1975), 215–225.

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

[44] 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.

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

[46] 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.

[47] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. 2020. Efficient bitruss decomposition for large-scale bipartite graphs. In *ICDE*. IEEE.

[48] Kai Wang, Shuting Wang, Xin Cao, and Lu Qin. 2020. Efficient radius-bounded community search in geo-social networks. *IEEE Transactions on Knowledge and Data Engineering* (2020).

[49] Nan Wang, Jingbo Zhang, Kian-Lee Tan, and Anthony KH Tung. 2010. On triangulation-based dense neighborhood graph discovery. *Proceedings of the VLDB Endowment* 4, 2 (2010), 58–68.

[50] Haiyuan Yu, Alberto Paccanaro, Valery Trifonov, and Mark Gerstein. 2006. Predicting interactions in protein networks by completing defective cliques. *Bioinformatics* 22, 7 (2006), 823–829.

[51] Long Yuan, Lu Qin, Wenjie Zhang, Lijun Chang, and Jianye Yang. 2017. Index-based densest clique percolation community search in networks. *IEEE Transactions on Knowledge and Data Engineering* 30, 5 (2017), 922–935.

[52] Si Zhang, Dawei Zhou, Mehmet Yigit Yildirim, Scott Alcorn, Jingrui He, Hasan Davulcu, and Hanghang Tong. 2017. Hidden: hierarchical dense subgraph detection with application to financial fraud detection. In *Proceedings of the 2017 SIAM International Conference on Data Mining*. SIAM, 570–578.