Disjoint k-Cliques in Real-World Graphs

Xin Chen
The Chinese University of Hong Kong
xchen@se.cuhk.edu.hk

Wenqing Lin
Tencent
edwlin@tencent.com

Haoxuan Xie
Nanyang Technological University
haoxuan001@e.ntu.edu.sg

Siqiang Luo
Nanyang Technological University
siqiang.luo@ntu.edu.sg

Sibo Wang
The Chinese University of Hong Kong
swang@se.cuhk.edu.hk

ABSTRACT

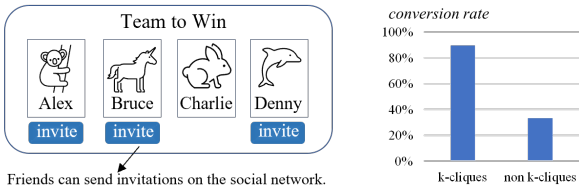
A k-clique is a dense graph, consisting of k fully-connected nodes, that finds numerous applications, such as community detection and network analysis. In this paper, we study a new problem, that finds a maximum set of disjoint k-cliques in a given large real-world graph with a user-defined number k, which can contribute to a good performance of teaming events in online games. However, this problem is NP-hard when $k \geq 3$, making it difficult to solve, not to mention that the real-world graph is often dynamic with frequent updates. To address that, we propose an efficient lightweight method that avoids the significant overheads and achieves a k-approximation to the optimal, which is equipped with several optimization techniques, including the ordering method, degree estimation in the clique graph, and a lightweight implementation. Besides, we devise an efficient indexing method with careful swapping operations that allows us to maintain a near-optimal result under sufficiently fast updates. In various experiments on several large graphs with dynamic settings, our proposed approaches significantly outperform the competitors by up to 2 orders of magnitude in running time and 13.3% in the number of computed disjoint k-cliques, which demonstrates the superiority of the proposed approaches in terms of efficiency and effectiveness.

PVLDB Reference Format:
Xin Chen, Wenqing Lin, Haoxuan Xie, Siqiang Luo, and Sibo Wang. Disjoint k-Cliques in Real-World Graphs. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/jerchenxin/disjoint-k-clique.

1 INTRODUCTION

Given an undirected graph G, a k-clique in G consists of exactly k nodes in G, each of which is incident to all the others. The k-clique is widely used in various applications, such as community detection [20, 24, 43] and social network analysis [12, 37]. In this paper, we study a new problem, *Disjoint k-Cliques*, which finds a maximum set of disjoint k-cliques, denoted by $\mathcal{S}$, in G for a fixed $k \geq 3$ such that (i) any two cliques in $\mathcal{S}$ do not share any common nodes, and (ii) the number of k-cliques in $\mathcal{S}$ is maximum. Note that, when k is 2, the problem is equivalent to maximum matching [14], whose solutions cannot be generalized to address the studied problem, which is NP-hard, as discussed in Section 3.



(a) A screenshot of the event (b) Team structures

Figure 1: A teaming event in a game

Applications. Computing the maximum set of disjoint k-cliques can be found in various real-world applications, especially in social networks. As an example, in a multiplayer online battle arena (MOBA) game of Tencent, which is one of the largest online game companies in China and has a massive number of users, the social network can be constructed based on the friendships made between players in the game. To enhance the participation of players, there are many teaming events using the social network in the game, as shown in Figure 1. Specifically, in these events, each user can join at most a team, whose capacity is k, and send invitations to friends in the team to win gaming rewards. For convenience, the game automatically generates all the teams and assigns each user to a team. We measure the performance of the event by the conversion rate, which is the ratio of users who win the rewards. As shown in Figure 1(b), users joining teams structured as k-cliques exhibit a conversion rate at least twice as high as those in teams not organized as k-cliques. The increased rate can be attributed to the dense structure of k-cliques, facilitating more effective communication among team members. Hence, to achieve a good performance of the events, finding such disjoint k-cliques as many as possible would be essential. Such a set of disjoint k-cliques can be found in many other applications. For example, in the problem of roommate allocation [30], where a room contains $k \geq 2$ beds, a good arrangement is to make the roommates in a room form as a k-clique in the graph constructed based on their preferences, which is equivalent to the computation of a maximum set of disjoint k-cliques.

A straightforward approach. To address this problem, a straightforward approach is to (i) first list all k-cliques in the graph, then (ii) construct a condensed graph, called *clique graph*, based on the k-cliques by taking each k-clique as a node and adding an edge between two nodes if the corresponding k-cliques are not disjoint, and (iii) finally adopt the algorithms for the maximum independent set (MIS) on the clique graph to compute the maximum set

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

of disjoint k-cliques. However, this approach suffers from several deficiencies that render it impractical for handling large graphs, explained as follows. Firstly, the number of k-cliques in a graph could be exponentially large. For example, as shown in Table 1, in the Facebook dataset consisting of 4 thousand nodes and 88 thousand edges, the number of 3-cliques is 1.61 million, which is at least 400 times as many as the number of nodes in the graph. Besides, the number of edges in the clique graph is at least 8.9 billion, making it highly dense. In other words, the size of the clique graph could be extremely large, especially when the original graph or k is sufficiently large. Secondly, computing the maximum independent set on large graphs is intensive [8]. In the experiments, we find that this approach can merely handle the graphs with only thousands of nodes, not to mention a larger k.

Our approaches. Instead of constructing the costly clique graph in the previous approach, our approaches do not need to materialize all the k-cliques, which significantly reduces the overheads of space consumption and running time. In particular, we develop a basic framework by starting from a node v in the graph G, and identifying a k-clique c incident to v. Afterwards, we remove all the nodes in c from G, as well as the corresponding edges, resulting in a residual graph G'. Note that, if there are not any k-cliques incident to v, we remove only v from G. We continue to identify the new k-clique in G', until the residual graph is empty. Therefore, we only need to maintain the k-cliques incident to the processed nodes and well prune the computation on the other nodes that have been in the chosen k-cliques, which largely narrows down the search space, rendering it possible to handle the large graphs widely existing in the real-world applications.

While the proposed framework is simple, there are several non-trivial techniques that can further improve the results in terms of both efficiency and effectiveness. Recall that we process the nodes in G sequentially, which poses a node ordering. As a result, the node ordering can greatly affect the performance of the proposed framework. To explain, consider that we process nodes in descending order of their degree in G, i.e., we start from the node with the largest degree among the unprocessed ones. Since a node v with a large degree might be associated with a large number of k-cliques, the process of node v would be able to prune a large portion of the search space due to disjointness. Furthermore, it might also result in the case where a small-degree node is difficult to be included in a k-clique, which might be pruned by the large-degree nodes, leading to the number of disjoint k-cliques being far from the maximum one. On the other hand, if we process the nodes in ascending order of their degree in G, the k-clique c computed on a small-degree node could include some large-degree nodes, making c incident to a massive number of the other k-cliques. In other words, this ordering could face the same issue as the previous ordering.

To address these issues, we propose to consider the node degree in the clique graph constructed in the aforementioned method, instead of the original graph. As such, we are able to identify a better ordering of nodes to generate the set of disjoint k-cliques by taking into account the relations between k-cliques, which is also adopted in the algorithms for MIS (see Section 3). However, computing the exact node degree in the clique graph is still costly, due to that it requires constructing the clique graph, whose size is too large to be processed. To alleviate this issue, we devise an approach to estimate the degree of each node in the clique graph efficiently and effectively. Specifically, we first calculate the number of k-cliques incident to each node in G by performing the k-clique listing algorithm in G without storing all the k-cliques. As a result, we are able to derive the estimation of degree in the clique graph for each k-clique c by exploiting the number of k-cliques incident to the nodes in c, as well as their neighbors in G, which also provide a lower bound and an upper bound of the degree in the clique graph. Based on that, we develop an efficient pruning strategy using the estimation of k-clique's degree in the clique graph, which enables us to look ahead at the nodes going to be processed, rendering it powerful in reducing the search space. Besides, we show that the proposed approach can achieve a k-approximation to the optimal solution, which guarantees its effectiveness. In the experiments, our proposed approach significantly outperforms the competitors in terms of both the quality of results and the required running time. For instance, in the Orkut dataset consisting of 3 million nodes and 117 million edges, compared to the competitors with $k = 6$, our proposed approach generates 13.3% more disjoint k-cliques and achieves a speedup by one order of magnitude.

Handling dynamic graphs. Nevertheless, real-world graphs often change frequently. For instance, the social network in the MOBA game of Tencent has a number of edge insertions or deletions in a day, caused by the construction or destruction of friendships between users, proportional to at least 1% of all edges in the graph. As a result, how to maintain the maximum set of disjoint k-cliques for dynamic graphs would be important, due to the requirements for both accuracy and efficiency in the aforementioned applications. To explain, the deletion of existing edges makes the computed results inaccurate, i.e., there could exist some k-cliques c in the existing results such that after deletion some nodes in c are not adjacent any longer, which means that c is not a k-clique in the updated graph. On the other hand, the insertion of new edges could affect the results by producing more disjoint k-cliques. To address this issue, a straightforward approach is to re-compute the maximum set of disjoint k-cliques on the updated graph. However, this approach is highly costly, making it impractical for real-world applications, due to that (i) computing the maximum set of disjoint k-cliques on a large graph takes a sufficiently large number of time, and (ii) the update of the graph could be frequent in the applications, which would require timely responses for queries. Therefore, we devise an efficient updating method that builds an indexing scheme for each k-clique c in the result set $\mathcal{S}$, which is a subset of k-cliques incident to c. As such, when dealing with an update affecting c, we can identify a replacement of c from the index and update the index accordingly, which is significantly faster than re-computing from scratch. Our dynamic approach is shown to be both efficient and effective based on experimental results. For instance, in the Orkut dataset, when k is 6, the average processing time for each update is just a few microseconds. Besides, the size of the updated $\mathcal{S}$ even increases slightly due to incurring more disjoint k-cliques.

Contributions. To sum up, we make the following contributions.

- We study the new problem of finding the maximum set of disjoint k-cliques in graphs, which has various applications in social networks. Besides, we show that this problem is NP-hard. (Section 2)

- We devise a lightweight method that avoids the significant overheads and achieves a k-approximation to the optimal, which is equipped with several optimization techniques, including the ordering method, degree estimation in the clique graph, and a lightweight implementation. (Section 4)
- We extend the proposed approaches to handle the dynamic graphs, which is required by real-world applications. We develop an efficient indexing method with careful swapping operations that allows us to maintain a near-optimal result under sufficiently fast insertion and deletion of edges. (Section 5)
- We demonstrate in various experiments that our proposed approach is able to handle large graphs with millions of nodes, and performs much better than several competitors in terms of both efficiency and effectiveness. In particular, our proposed approach achieves up to 13.3% more k-cliques and consumes up to two orders of magnitude less running time than the competitors, making it suitable to handle large graphs. Besides, we show that our approaches can handle dynamic graphs efficiently and obtain a high-quality result simultaneously. (Section 6)

2 PRELIMINARIES

In this section, we introduce the basic concepts and the problem definition. Let $G = (V, E)$ be an undirected graph, where V is the set of nodes with cardinality $n = |V|$, E is the set of edges with cardinality $m = |E|$, and each edge connecting nodes u and v is denoted by $\langle u, v\rangle$. We first define the core concept, *k-clique*, as follows.

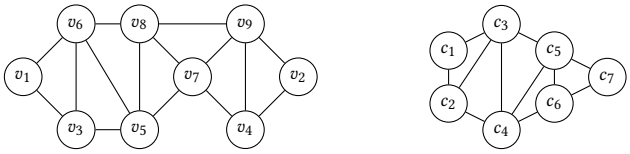
Definition 2.1 (k-Clique). A k-clique c is a graph with exactly k nodes, denoted by $c = (u_1, u_2, ..., u_k)$, each of which is incident to all the others in the graph. That is, we have $\langle u, u'\rangle \in E$ for any $u \in c$ and $u' \in c$.

Note that, k is a user-defined parameter and should be no less than 3. We say that two k-cliques c_1 and c_2 are *disjoint*, if there does not exist a node v that exists in both c_1 and c_2. Therefore, we have the definition of *disjoint k-clique set*, stated in the following.

Definition 2.2 (Disjoint k-Clique Set). A disjoint k-clique set $\mathcal{S}$ is a set of k-cliques such that each clique in $\mathcal{S}$ is disjoint with all other cliques in $\mathcal{S}$.

The size of a disjoint k-clique set $\mathcal{S}$ is defined as the number of k-cliques in $\mathcal{S}$. We say that $\mathcal{S}$ is *maximal* if we can not add any other k-clique in G into $\mathcal{S}$ without violating the disjoint constraint. Furthermore, we say that $\mathcal{S}$ is *maximum* if its size is the largest among all disjoint k-clique sets in G. Note that, a maximum disjoint k-clique set is a maximal one and not unique, i.e., there could be several different disjoint k-clique sets of the largest size.

Example 2.3. In Figure 2(a), there is a graph G with 9 nodes and 15 edges. Consider k is 3. The number of 3-cliques is 7, listed as follows: (v_1, v_3, v_6), (v_3, v_5, v_6), (v_5, v_6, v_8), (v_5, v_7, v_8), (v_7, v_8, v_9), (v_4, v_7, v_9), and (v_2, v_4, v_9). There could be several disjoint 3-clique sets, one of which could include two 3-cliques (v_3, v_5, v_6) and (v_4, v_7, v_9), denoted by $\mathcal{S}_1$, as these cliques share no nodes in common. Note that, $\mathcal{S}_1$ is maximal, as we can not add any other cliques to $\mathcal{S}_1$ without violating the constraint of disjointness. Besides, a maximum disjoint 3-clique set includes three 3-cliques (v_1, v_3, v_6), (v_5, v_7, v_8), and (v_2, v_4, v_9).



(a) G contains seven 3-cliques (b) Clique graph G_C

Figure 2: G and its clique graph G_C

Finding the maximum set of disjoint k-cliques is extremely costly, which can be explained in the following theorem. All proofs can be found in the full-version technical report at https://github.com/jerchenxin/disjoint-k-clique/blob/main/technical_report.pdf.

Theorem 2.4. *The maximum disjoint k-clique problem is NP-hard when k is fixed and no less than 3.*

Proof. When k is 2, this problem equals the maximum matching, which can be solved in polynomial time [14]. As such, we only consider the case where k is no less than 3. In such case, we prove its hardness by reducing the maximum matching in k-uniform hypergraphs, which is NP-hard [27, 32, 39], to the maximum disjoint k-clique problem. The maximum matching in k-uniform hypergraphs finds a set of hyperedges with the largest size where each hyperedge consists of exact k nodes and no two hyperedges share common nodes. Suppose there is a k-uniform hypergraph $G_H = (V_H, E_H)$, we can build a new graph $G = (V, E)$ by regarding each hyperedge $\langle v_1, v_2, ..., v_k\rangle$ as a k-clique in G. In detail, for each hyperedge, we add its nodes to V and add an edge $\langle u, v\rangle$ to E between any two nodes u and v in this hyperedge. Building G costs $O(|E_H| \cdot \binom{k}{2})$ time, which is in polynomial time when k is fixed. As a consequence, the maximum disjoint k-clique problem is equivalent to the problem of maximum matching in k-uniform hypergraphs, which means that the maximum disjoint k-clique problem is NP-hard. □

Due to its hardness, in this paper, we focus on finding a near-optimal maximal disjoint k-clique set, denoted by $\mathcal{S}$, when k is fixed and no less than 3, in both static and dynamic graphs. The formal problem statement is defined as follows.

Problem statement. We study the problem of computing the maximal disjoint k-cliques in both static and dynamic graphs. Specifically, given an undirected graph $G = (V, E)$ and a user-defined parameter $k \geq 3$, we aim to compute a near-optimal maximal disjoint k-clique set $\mathcal{S}$. Furthermore, when G is updated with edge insertion or deletion, we aim to maintain $\mathcal{S}$ as a near-optimal maximal set on the updated graph.

3 RELATED WORK

In this section, we review the related problems, listing k-cliques, the maximum independent set, the maximum matching, and dynamic graph methods.

Listing k-cliques. There are numerous studies on k-clique listing [9, 11, 34, 40, 48, 49]. Among these, we introduce the k-clique listing framework proposed by Danisch et al. [11] in detail as it is the first parallel k-clique listing framework and our approaches apply this framework in our frequent k-clique listing operations. This work proposes a parallel k-clique listing method with two parallel methods (node-parallel and edge-parallel). Here, we only review

the node-parallel method, which is more efficient when k is small in practice. To explain, when k is small, the benefits brought by the fine-grained parallelism of the edge-parallel method cannot outweigh its cost of constructing more computation graphs.

Algorithm 1 presents the details of the k-clique listing method. Algorithm 1 first converts the original graph G into a directed acyclic graph (DAG) (Lines 1-2). In detail, given an arbitrary total node ordering, each edge in G adds a direction from the high-order node to the low one. For the choice of the total node ordering, please refer to [40] for an extensive study. DAG can help avoid the redundant clique computation. To explain, the k-clique set in the subgraph induced by a node u_1 and its out-neighbors will have no common k-clique with the k-clique set in another subgraph induced by any other node u_2 and its out-neighbors. Here, the subgraph induced by a set of nodes is generated in the following steps: (i) add all nodes of this set to this subgraph, and (ii) add an edge between two nodes if there exists an edge connecting these two nodes in the original graph. For example, in Figure 3, the subgraph induced by a node v_6 and its out-neighbors contains two 3-cliques: (v_1, v_3, v_6) and (v_3, v_5, v_6) while the subgraph induced by a node v_8 and its out-neighbors contains two 3-cliques: (v_5, v_6, v_8) and (v_5, v_7, v_8). We can observe that these two k-clique sets contain no common k-clique. With this property, we can parallel list k-cliques of the subgraph induced by each node and its out-neighbors. We use the notation $\vec{G}$ to denote a directed graph, $N^+(u)$ to denote the set of out-neighbors of u in $\vec{G}$, and $\vec{G}[S]$ to denote the induced subgraph by the set S of nodes. After building a DAG, it calls the procedure *Listing* for each node u in the subgraph $\vec{G}[N^+(u)]$ in a node parallel paradigm (Lines 3-4). Given an integer l, a subgraph $\vec{G}$, and a set C of nodes, the procedure *Listing* finds all l-cliques in the subgraph $\vec{G}$ and then unions each l-cliques with C and finally get all k-cliques. Remarkably, nodes in l-cliques must be incident to nodes in C as the subgraph $\vec{G}[N^+(u)]$ is induced by the out-neighbors of u. Thus, finally, we can union nodes in l-cliques with nodes in C to form k-cliques. When l is 2, a l-clique is actually an edge. Thus, each edge in $\vec{G}$ together with C forms a k-clique (Lines 7-9). Otherwise, it iteratively calls *Listing* for each node u in $\vec{G}$ with a new subgraph $\vec{G}[N^+(u)]$ (Lines 10-12). Notably, when the input graph $\vec{G}$ is empty or contains no edges, it will not output k-cliques as it can not reach Line 9. The time complexity is $O(k \cdot m \cdot (\frac{cv(G)}{2})^{k-2})$ [11], where $cv(G)$ is the core value [3] of the graph which is defined as the maximum integer i such that there exists an induced subgraph of G with all nodes having degree at least i. Notably, these algorithms for listing k-cliques can not address our studied problem, since they do not consider the disjoint constraint between cliques in $\mathcal{S}$.

Algorithm 1: kCliqueListing(G)

Input: An undirected graph $G = (V, E)$, k
Output: All k-cliques
1 Let η be a total ordering on $V(G)$
2 $\vec{G} \leftarrow$ a directed version of G by retaining the direction $u \rightarrow v$ for each edge $(u, v) \in E$ if $\eta(u) > \eta(v)$
3 **for each** *node* $u \in V(G)$ *in parallel* **do**
4 $\quad$ *Listing*$(k-1, \vec{G}[N^+(u)], \{u\})$
5
6 **Procedure** *Listing*$(l, \vec{G}, C)$
7 **if** $l = 2$ **then**
8 $\quad$ **for each** *edge* $\langle u, v\rangle$ *of* $\vec{G}$ **do**
9 $\quad\quad$ **output** k-clique $C \cup \{u, v\}$
10 **else**
11 $\quad$ **for each** *node* $u \in V(\vec{G})$ **do**
12 $\quad\quad$ *Listing*$(l-1, \vec{G}[N^+(u)], C \cup \{u\})$

MIS. Existing solutions for MIS can be categorized into three folds: exact algorithms [2, 17, 46], approximate algorithms [5, 22], and heuristic algorithms [8, 40]. As the recent heuristic algorithms can efficiently produce near-optimal solutions, we review the reducing-peeling framework [8] used in sparse graphs and another heuristic method [35, 40] used in dense graphs.

For sparse graphs, the reducing-peeling framework works well. It follows a strategy of applying reduction rules to nodes in order to reduce the graph size while maintaining the maximum independent set. The reduction rules are continuously applied until no further reduction is possible. In cases where no reduction rules can be applied, the framework utilizes a peeling strategy to break ties. In terms of efficiency, it is preferable to apply reduction rules to nodes with small degrees. We illustrate the degree-one reduction rule as an example. Suppose that there exists an edge $\langle u, v\rangle$ where the degree of u is 1. In such a case, we can safely add u to the independent set and remove u and v from G without affecting the optimal independent set. Effective peeling strategies include directly removing a high-degree node from G. However, this framework can not be applied in clique graphs as the clique graphs are dense and there are nearly no small degree nodes to reduce, resulting in numerous costly peeling operations.

For dense graphs, an effective greedy method [35, 40] involves iteratively adding the minimum-degree node to an initially empty solution while simultaneously removing this node and its neighbors from the graph until the graph is empty. However, this method still can not address our studied problem since even calculating the node degree in the clique graph is computationally expensive.

Maximum matching in k-uniform hypergraphs. The maximum matching in k-uniform hypergraphs finds a set of hyperedges with the largest size where each hyperedge consists of exact k nodes and no two hyperedges share common nodes. There exists a plethora of methods [4, 6, 7, 10, 13, 16, 18, 21, 23, 25, 29, 42] for finding a maximum matching in k-uniform hypergraphs. Among these methods, numerous studies aim to improve the approximate ratio. For example, [29] presented a $(\frac{k}{2} + \epsilon)$-approximation method; [21] established a quasi-polynomial $(\frac{k}{3})$-approximation method; [42] provided a polynomial time $(\frac{k+2}{3})$-approximation method; [10, 18] presented a polynomial time $(\frac{k+1}{3}+\epsilon)$-approximation method. Noteworthy, methods for the maximum matching in k-uniform hypergraphs can not be directly applied in our disjoint k-clique problem as these methods assume that every hyperedge is listed in advance. However, listing all cliques in our problem in advance requires huge memory overheads leading to out-of-memory issues.

Maximum matching in general unweighted graphs. We briefly review the maximum matching in unweighted graphs, which finds

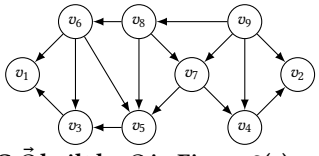


Figure 3: A DAG $\vec{G}$ built by G in Figure 2(a) using a total node ordering, $\{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9\}$.

a maximum set of edges where each edge contains no common node with others. This problem can be solved in polynomial time [14]. There exist numerous works [14, 15, 19, 31, 36] for improving the time complexity of this problem. [14] proposed a method that runs in $O(n^4)$ by shrinking *Blossom* [14]. [19], [15], and [36] further improved the blossom-based method to $O(n^3)$, $O(n^{2.5})$, and $O(\sqrt{n} \cdot m)$, respectively. Instead of using the blossom, [31] introduced methods using *Augmenting Path* [31], which run in $O(n \cdot m)$. Particularly, this problem is a special case of our studied problem when k is 2. Thus, it can not address our studied problem, which is a more generalized case.

Dynamic graph methods. There are numerous approaches on dynamic graph for various problems, such as shortest path queries [26, 50, 51], community search [1, 28, 47], and reachability queries [38, 45, 52]. However, these problems are orthogonal to the studied new problem, making their solutions unable to compute the maximum set of disjoint k-cliques efficiently.

4 PROPOSED ALGORITHMS

In this section, we first present a basic framework that avoids the overheads in the clique graph construction. After that, we illustrate the optimization techniques based on the framework to generate a near-optimal result efficiently.

4.1 A Basic Framework

As discussed in Section 1, a straightforward approach is to construct the clique graph and then apply the algorithms for MIS. However, this approach is highly costly, rendering it impossible to handle sufficiently large graphs. To address this issue, we propose a basic framework that does not need to construct the expensive clique graph, which significantly reduces the overheads. In particular, the basic framework begins with an empty set $\mathcal{S}$. Then, it works in several iterations. In each iteration, we add a k-clique c in G to $\mathcal{S}$ if c is disjoint with the k-cliques in $\mathcal{S}$, and remove all nodes in c from G. The iteration stops when the residual graph of G is empty or has no more k-clique that can be added into $\mathcal{S}$. Algorithm 2 presents the details of the basic framework on G. It first converts G into a DAG (Line 3) using a total node ordering. Similar to Algorithm 1, using a DAG can avoid redundant clique computation. Besides, it can improve the quality of $\mathcal{S}$. To explain, suppose that we use a degree ordering. For each processed node u, its computation graph $\vec{G}[N^+(u)]$ only includes nodes with degrees no larger than u. Thus, after adding a clique including u to $\mathcal{S}$ and removing nodes of this clique from G, the search space only decreases slightly, where the search space means the number of cliques in the remaining graph. Besides, every node u maintains a value $valid(u)$ (Line 4), which records whether this node is included in some clique of $\mathcal{S}$. Then, for each node u in ascending order, it calls the procedure

Algorithm 2: BasicFramework(G)

```
   Input: An undirected graph G, k
   Output: A set S of disjoint k-cliques
1  S ← ∅
2  Let η be a total ordering on V(G)
3  G⃗ ← directed version of G, where u → v if η(v) < η(u)
4  valid(v) ← false ∀v ∈ V
5  for each node u in ascending order of η(u) do
6      if valid(u) = true and |N+(u)| ≥ k − 1 then
7          if FindOne(k − 1, G⃗[N+(u)], {u}) = ture then
8              for each node v in the newly found clique C do
9                  valid(v) ← false
10                 Remove v from G⃗
11             S ← S ∪ C
12 return S
13
14 Procedure FindOne(l, G⃗, C)
15 if l = 2 then
16     Find one edge ⟨u, v⟩ of G⃗ and form a k-clique C ∪ {u, v}
17     return true
18 else
19     for each node u ∈ V(G⃗) do
20         if |N+(u)| < l − 1 then
21             continue
22         if FindOne(l − 1, G⃗[N+(u)], C ∪ {u}) = ture then
23             return true
24 return false
```

FindOne to find a $(k-1)$-clique in the subgraph $\vec{G}[N^+(u)]$ that is first encountered (Lines 5-11). Then, the first encountered $(k-1)$-clique and node u form a k-clique. Given a parameter l, *FindOne* returns immediately once it finds a l-clique (Lines 17 and 23). After *FindOne* returns a clique c, it directly adds c into $\mathcal{S}$ and updates the graph $\vec{G}$ by removing nodes of this clique from $\vec{G}$ (Lines 8-11).

Example 4.1. Figure 3 shows an example of Algorithm 2 when k is 3. The node ordering is $\{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9\}$. We build a DAG $\vec{G}$ using this ordering from G in Figure 2(a). First, we deal with $\vec{G}[N^+(v_6)]$ as there is no 2-clique in $\vec{G}[N^+(v_1)]$, $\vec{G}[N^+(v_2)]$, $\vec{G}[N^+(v_3)]$, $\vec{G}[N^+(v_4)]$, and $\vec{G}[N^+(v_5)]$. Though there are two cliques (v_1, v_3, v_6) and (v_3, v_5, v_6) in $\vec{G}[N^+(v_6)]$, suppose we first find (v_3, v_5, v_6), we directly add this clique to $\mathcal{S}$, and delete these three nodes from $\vec{G}$. As there is no 2-clique in $\vec{G}[N^+(v_7)]$ and $\vec{G}[N^+(v_8)]$, we turn to $\vec{G}[N^+(v_9)]$ and find three cliques (v_7, v_8, v_9), (v_4, v_7, v_9), and (v_2, v_4, v_9). Suppose we first find (v_7, v_8, v_9), we then directly add this clique to $\mathcal{S}$, and delete these three nodes from $\vec{G}$. It terminates after all nodes have been processed, and $\mathcal{S}$ contains two 3-cliques, (v_3, v_5, v_6) and (v_7, v_8, v_9).

Analysis. The quality of $\mathcal{S}$ produced by Algorithm 2 depends on the selection of k-cliques to be added into $\mathcal{S}$, which has two key aspects related to it. First, the traversal order in each computation graph $\vec{G}[N^+(u)]$ matters. In Example 4.1, when dealing with $\vec{G}[N^+(v_6)]$,

if we first find (v_1, v_3, v_6) instead of (v_3, v_5, v_6), it will produce a maximum $\mathcal{S}$. Second, a good total node ordering enables k-cliques in the computation graph to slightly impact the search space as mentioned above. Though a good total node ordering can alleviate the problem of the selection of k-cliques, it can not address it well as it still suffers from the problem in the first case. Nevertheless, Algorithm 2 is efficient as its time complexity is $O(k \cdot m \cdot (\frac{cv(G)}{2})^{k-2})$. To explain, it lists every k-clique at most once, and thus, in the worst case, the time complexity is the same as that of Algorithm 1. Its space complexity is $O(m + n + |\mathcal{S}|)$.

4.2 k-Clique Ordering

As aforementioned, the selection of k-cliques to be added into $\mathcal{S}$ could significantly affect the size of $\mathcal{S}$. To address this issue, we propose an ordering of k-cliques of G based on the node degree estimation of the clique graph, which would be able to generate a large number of disjoint k-cliques. Note that, the estimation does not need to construct the clique graph, which could avoid the overheads as mentioned previously. For ease of illustration, we first introduce the definitions of clique graph and the related concepts, and then explain the methodology for estimation with a theoretical guarantee.

Definition 4.2 (Clique Graph). Given all k-cliques in G, the clique graph of G, denoted by G_C, is an undirected graph, where (i) each k-clique of G is a node in G_C, and (ii) there is an edge connecting any two nodes in G_C if the corresponding k-cliques are not disjoint.

We say that two k-cliques c_1 and c_2 are *neighbors* in the clique graph G_C if there exists a node u such that u appears in both c_1 and c_2. In other words, c_1 and c_2 are not disjoint. Based on that, we have the *degree* of a k-clique, defined as follows.

Definition 4.3 (Clique Degree). Given a clique graph G_C and a clique c, the clique degree of c, denoted by $deg_{G_C}(c)$, equals the number of neighbors of c in G_C.

Recall that a straightforward approach is to compute the MIS on the clique graph. While this approach is infeasible, especially on large graphs, it can lead to a near-optimal result by based on the algorithms for MIS, which consider the degree of k-cliques in the clique graph, as explained in Section 3. However, to compute the exact degree of each k-clique in the clique graph is extremely costly, due to that it requires the construction of the clique graph.

To address this issue, we devise a scoring method that effectively approximates the degree of each k-clique in the clique graph. Specifically, we first develop a method to reflect the number of k-cliques containing the nodes in G, as follows.

Definition 4.4 (Node Score). Given a node u in G, the node score of u, denoted by $s_n(u)$, is the number of k-cliques containing u.

Therefore, the score of a k-clique c can be computed based on the scores of nodes in c, which is defined in the following.

Definition 4.5 (Clique Score). Given a clique c, the clique score of c, denoted by $s_c(c)$, equals the total score of nodes in c, i.e., $\sum_{u \in c} s_n(u)$.

To further illustrate the above definitions, we provide the following examples.

Example 4.6. In Figure 2(a), when $k = 3$, we have seven 3-cliques in G, i.e., (v_1, v_3, v_6), (v_3, v_5, v_6), (v_5, v_6, v_7), (v_5, v_7, v_8), (v_7, v_8, v_9), (v_4, v_7, v_9), and (v_2, v_4, v_9), denoted as $c_1, c_2, \cdots, c_7$ respectively. Therefore, we can construct the clique graph G_C of G, as shown in Figure 2(b). Two nodes in G_C have an edge only if the two corresponding cliques have common nodes. For example, c_1 and c_2 share the common node v_3, resulting in an edge $\langle c_1, c_2 \rangle$ connecting c_1 and c_2. Therefore, the clique degree of each 3-clique can be computed in G_C, e.g., $deg_{G_C}(c_1) = 2$ due to that c_1 is incident to two 3-cliques c_2 and c_3 in G_C. Furthermore, we have the node score of v_6 as $s_n(v_6) = 3$, since there are three 3-cliques containing v_6, which are (v_1, v_3, v_6), (v_3, v_5, v_6), and (v_5, v_6, v_8). Similarly, we have $s_n(v_5) = 3$ and $s_n(v_8) = 3$. Moreover, since the 3-clique c_3 contains three nodes v_5, v_6 and v_8, we have the clique score of c_3 as $s_c(c_3) = s_n(v_5) + s_n(v_6) + s_n(v_8) = 9$.

As such, we can derive an upper bound and a lower bound of the clique degree for each k-clique c in G based on the clique score of c, which is stated as follows.

Theorem 4.7. *Given a k-clique c, the degree $deg_{G_C}(c)$ of c in G_C satisfies that $(s_c(c) - k)/(k-1) \leq deg_{G_C}(c) \leq s_c(c) - k$.*

Proof. Consider a k-clique c, which contains k nodes, denoted by $v_1, v_2, ..., v_k$. For each node u in c, let $S(u)$ be the set of k-cliques containing u. Then, we have $c \in S(u)$ and the size of $S(u)$ is $s_n(u)$. As each k-clique c' in $S(u)$, excepting c, should be a neighbor of c in the clique graph due to that they have a common node u with c, the contribution of $S(u)$ to the degree of c is at most $s_n(u)-1$. Therefore, we have an upper bound of the degree of c by taking into account the node scores of nodes in c, i.e., $deg_{G_C}(c) \leq \sum_{u \in c}(s_n(u) - 1) = s_c(u) - k$, according to Definition 4.5. On the other hand, since each k-clique c' in $S(u)$ has $k-1$ nodes excepting u, meaning that c' might be incident to $k-1$ other k-cliques which is not c, the contribution of $S(u)$ to the degree of c is at least $(s_n(u) - 1)/(k-1)$. As a result, we obtain a lower bound of $deg_{G_C}(c)$ as $\sum_{u \in c}(s_n(u) - 1)/(k-1) = (s_c(u) - k)/(k-1)$, which completes the proof. □

In other words, the degree of a k-clique c is highly related to its clique score $s_c(c)$, which means that we can approximate the degree of c in the clique graph by $s_c(c)$. Consequently, we can obtain an approximate degree of each k-clique in G without explicitly constructing the clique graph, which would incur significant overheads in the computation and memory consumption. To achieve that, we propose an algorithm that utilizes the clique score to generate a near-optimal result set $\mathcal{S}$.

Algorithm 3 presents the details of our proposed approach, which first collects all k-cliques (Line 2) and then processes the k-cliques in the ascending order of their clique scores in several iterations (Lines 3-5). In each iteration, we add a k-clique c into $\mathcal{S}$, if c is disjoint with all k-cliques in $\mathcal{S}$. The algorithm terminates when all k-cliques are processed and returns $\mathcal{S}$ as the result.

Analysis. In the sequel, we first provide that the proposed algorithm can achieve a k-approximation to the optimal result. After which, we analyze the complexity of the proposed algorithm in terms of both space consumption and running time.

Algorithm 3: ComputeWithCliqueScores(G)

Input: An undirected graph G, k
Output: A set $\mathcal{S}$ of disjoint k-cliques

```
1 S ← ∅
2 List and collect all k-cliques of G, calculate s_c(c) for each clique c
3 for each clique c in the ascending order of s_c(c) do
4     if c is disjoint with all k-cliques in S then
5         S ← S ∪ c
6 return S
```

Lemma 4.8. *For a clique graph G_C and a node c in G_C, there exist two neighbors among any $k+1$ neighbors of c in G_C such that these two neighbors are connected with an edge in G_C.*

Proof. Suppose that there is a k-clique c. Any neighbor of c must share at least one node of c_i. Thus, for its $k+1$ neighbors, there are at least two neighbors that share the same node of c, indicating there is an edge connecting these two neighbors in G_C. □

Theorem 4.9. *Any maximal $\mathcal{S}$ is a k-approximation solution.*

Proof. Suppose that we have a maximal disjoint k-clique set $\mathcal{S}$, by Lemma 4.8, when we remove a k-clique from $\mathcal{S}$, we can add up to k of its neighbors to $\mathcal{S}$ without violating the disjoint constraint. Thus, the approximate ratio is k. □

Algorithm 3 can produce a near-optimal solution as every clique is chosen from a nearly global optimal view. However, this method is not efficient as it has to collect all cliques. Its time complexity is $O(k \cdot m \cdot (\frac{cv(G)}{2})^{k-2} + C \cdot \log C)$, where C is the number of all cliques. To explain, it needs to find a clique with the minimum clique score in each iteration, which requires an additional cost of $O(C \cdot \log C)$. Its space complexity is $O(m + n + |\mathcal{S}| + C)$ as it needs to store all k-cliques.

4.3 A Lightweight Implementation

The drawback of Algorithm 3 is that it requires computing and storing all the k-cliques in G in memory, which results in significant overheads in memory consumption. To explain, consider the Facebook dataset discussed in Section 1, whose 3-cliques have a number at least 400 times than the one of nodes. Therefore, storing all the k-cliques would explode the memory space, especially for large graphs or a sufficiently large k. To alleviate this issue, we develop a lightweight implementation that (i) does not need to store all the k-cliques, and (ii) produces the same result as the one of Algorithm 3.

Algorithm 4 presents the proposed lightweight implementation. The main idea is first to find a k-clique with the local minimum clique score for the subgraph $\vec{G}[N^+(u)]$ of each node u. Then, these previously found k-cliques are collected, and a clique with the global minimum clique score among them is found in each iteration. In detail, it first calculates the node score for each node (Line 2) and then sets a total node ordering using the node score (Line 3). After that, it converts the original graph into a DAG using the total ordering (Line 4). To efficiently find the clique with the global minimum clique score, it uses a min-heap *MinHeap* to maintain

Algorithm 4: Lightweight(G)

Input: An undirected graph G, k
Output: A set $\mathcal{S}$ of disjoint k-cliques

```
1  S ← ∅
2  List and store all k-cliques of G, calculate s_c(c) for each clique c
3  Let η be a total ordering on V(G) such that if η(u) < η(v), then
     s_n(u) ≤ s_n(v)
4  G⃗ ← a directed version of G, where u → v if η(v) < η(u)
5  MinHeap ← ∅, valid(v) ← false ∀v ∈ V
6  HeapInit(MinHeap)
7  Calculation(MinHeap, S)
8  return S
9
10 Procedure HeapInit(MinHeap)
11 for each node u ∈ V(G⃗) in parallel do
12     if |N+(u)| ≥ k − 1 then
13         FindMin(l − 1, G⃗[N+(u)], {u}, ∅, s_n(u))
14         Push the output clique into MinHeap
15
16 Procedure FindMin(l, G⃗, C, C_min, S_cur)
17 if l = 2 then
18     for each node u of V(G⃗) do
19         if S_cur + s_n(u) ≥ s_c(C_min) then
20             continue
21         for each edge ⟨u, v⟩ of G⃗ do
22             C_new ← C ∪ {u, v}
23             if s_c(C_new) < s_c(C_min) then
24                 C_min ← C_new
25 else
26     for each node u ∈ V(G⃗) do
27         if |N+(u)| < l − 1 or S_cur + s_n(u) ≥ s_c(C_min) then
28             continue
29         FindMin(l − 1, G⃗[N+(u)], C ∪ {u}, minC, S_cur + s_n(u))
30
31 Procedure Calculation(MinHeap, S)
32 while MinHeap is not empty do
33     Pop a clique c where node u has the largest node ordering
34     if c is disjoint with all k-cliques in S then
35         Repeat Lines 8-11 of Algorithm 2
36     else
37         if valid(u) = true and |N+(u)| ≥ k − 1 then
38             FindMin(l − 1, G⃗[N+(u)], {u}, ∅, s_n(u))
39             Push the output clique into MinHeap
```

cliques with the local minimum clique score (Line 5). Then, it calls the procedure *HeapInit* to initialize *MinHeap* (Line 6), which calls the procedure *FindMin*, for each node u, to find the clique with the minimum clique score in the subgraph $\vec{G}[N^+(u)]$ and pushes this clique into *MinHeap* (Lines 11-14). After initializing *MinHeap*, it calls the procedure *Calculation* to calculate $\mathcal{S}$ by finding a clique with the global minimum clique score in each iteration (Line 7). This procedure pops cliques from *MinHeap* until *MinHeap* is empty (Lines 32-39). During this step, when a clique is disjoint with all

k-cliques in $\mathcal{S}$, then it adds this clique to $\mathcal{S}$ and updates $\vec{G}$ (Lines 34-35). Otherwise, if its internal node u with the highest node ordering is still valid, it finds and pushes a new clique with the minimum clique score in the subgraph $\vec{G}[N^+(u)]$ to *MinHeap* (Lines 37-39).

We implement our score-driven pruning strategy in the procedure *FindMin*. The motivation behind this strategy is to prune certain branches when the sum of the node scores of the previous recursive nodes is no smaller than the minimum clique score that has been found, indicating such branches can not produce a clique with a smaller clique score. In detail, *FindMin* includes a parameter S_{cur}, which represents the sum of the node score of the previous recursive node nodes. With this, we implement our score-driven pruning strategy (Lines 19-20 and Lines 27-28). Specifically, a branch is pruned when the value of S_{cur} plus the score of the currently visited node is no smaller than the minimum clique score that has been found.

Analysis. With converting G into a DAG $\vec{G}$, the k-clique sets of the subgraph $\vec{G}[N^+(u)]$ for each node u are disjoint, and their union exactly corresponds to the entire k-clique set. Based on this observation, we can first find the clique with the local minimum clique score of the subgraph $\vec{G}[N^+(u)]$ for each node u in parallel and then identify the clique with the global minimum score among these found cliques. When an invalid clique from the subgraph $\vec{G}[N^+(u)]$ is popped, we should update the clique with the local minimum clique score in $\vec{G}[N^+(u)]$, which incurs redundant computation. However, such redundant computation must happen with some nodes becoming invalid and removed in the subgraph $\vec{G}[N^+(u)]$, which indicates the size of this subgraph gradually reduces. Furthermore, together with our score-driven pruning strategy, such redundant computation is limited. The time complexity of the algorithm is $O(n \cdot m \cdot (\frac{cv(G)}{2})^{k-2})$. To explain, Algorithm 4 involves redundant k-clique listing computation where for each node u, it conducts computation in its subgraph $\vec{G}[N^+(u)]$ at most n/k times. The space complexity is $O(m + n + |\mathcal{S}|)$.

Theorem 4.10. *Given a fixed total node ordering and a fixed total ordering between cliques, Algorithm 4 and Algorithm 3 produce the same $\mathcal{S}$.*

Proof. A fixed total node ordering guarantees that $\vec{G}$ of two algorithms are the same. A fixed total ordering between cliques guarantees a fixed ordering between any two cliques with the same clique score. As they always find the k-clique with the minimum clique score in each iteration, they produce the same $\mathcal{S}$. □

5 HANDLING DYNAMIC GRAPHS

Previously, we assume that the graph is static, while most real-life graphs are dynamic with frequent updates by inserting new edges or deleting existing edges, as observed in Section 1. Note that, the case of updates on the nodes can be treated equivalently as the updates on the edges incident to the corresponding nodes. Handling updates efficiently for our studied problem would be of great importance, due to that (i) some k-cliques incident to deleted edges could no longer exist in the updated graph, (ii) the result set $\mathcal{S}$ might no longer be the largest one, especially after inserting a sufficient number of edges, and (iii) the real-world applications would require a timely response for the updates.

Algorithm 5: TrySwap(q)

Input: A directed graph $\vec{G}$, k, a Queue q of cliques in $\mathcal{S}$
Output: A set $\mathcal{S}$ of disjoint k-cliques

```
1 while q is not empty do
2     c ← q.pop()
3     Locate all neighboring candidate cliques C(c) of c
4     Use Algorithm 3 to find disjoint k-cliques S_dis among C(c)
5     if |S_dis| > 1 then
6         Remove c from S and add cliques in S_dis to S
7         Update the candidate cliques
8         Push any cliques c' in S whose C(c') involves new
            candidate cliques to q
9 return S
```

To achieve that, we propose an efficient strategy by first inspecting a sufficiently small set S of k-cliques that would be affected by the updates, and then swapping k-cliques in $\mathcal{S}$ with the ones in S that would maximize the size of the result set $\mathcal{S}$. In order to make the inspection efficient, we develop an indexing approach that can easily retrieve the set S according to the updates. In the following, we first present the algorithm designed for efficient swap operations in Section 5.1, and then we introduce the indexing structures in Section 5.2, after which we put all together by explaining the algorithms for handling the updates based on the swapping operations and the indexing structures in Section 5.3.

5.1 Swap Operations

The main idea behind our swap operation is to remove a clique from $\mathcal{S}$ and add as many of its disjoint neighboring cliques as possible to $\mathcal{S}$ in order to increase the size of $\mathcal{S}$. For instance, in Figure 4(b), suppose that $\mathcal{S}$ initially contains two 3-cliques (v_9, v_{10}, v_{11}) and (v_3, v_4, v_5). Then, we can remove the clique (v_3, v_4, v_5) from $\mathcal{S}$, add its two disjoint neighboring cliques (v_1, v_2, v_3) and (v_5, v_6, v_7) to $\mathcal{S}$. By doing so, we can increase the size of $\mathcal{S}$ by 1. The swap operations aim to optimize the size of $\mathcal{S}$ by iteratively applying a swap in a clique of $\mathcal{S}$, thereby maximizing the size of $\mathcal{S}$.

From the above example, we can see that the clique (v_4, v_9, v_{11}) is also the neighboring clique of (v_3, v_4, v_5). However, it has no opportunity to be added to $\mathcal{S}$ with the removal of (v_3, v_4, v_5) since it is also the neighboring clique of another clique (v_9, v_{10}, v_{11}), which is also in $\mathcal{S}$, and adding it to $\mathcal{S}$ will violate the disjoint constraint. From this observation, when applying the swap in a clique, we actually do not need to list all neighboring cliques, which is an expensive k-clique listing operation. Thus, we first define *Candidate k-Clique* to accelerate the swap operation.

Definition 5.1 (Free Node). Given a set $\mathcal{S}$ of disjoint k-cliques, we say a node u is a free node if it is not included in any clique of $\mathcal{S}$.

Definition 5.2 (Candidate k-Clique). Given a set $\mathcal{S}$ of disjoint k-cliques, we say a k-clique c_i in G is a candidate k-clique only if it has at least one free node and any internal node is either a free node or involved in a certain k-clique c_j in $\mathcal{S}$, i.e., non-free nodes are all involved in this k-clique c_j.

We use C to denote all candidate k-cliques and $C(c)$ to denote a set of candidate k-cliques of a clique c in $\mathcal{S}$, where each candidate

Algorithm 6: CONSTRUCTION($\mathcal{S}$)

Input: A directed graph $\vec{G}$, k, A set $\mathcal{S}$ of disjoint k-cliques
Output: A set C of candidate k-cliques
1 **for each** *clique c in $\mathcal{S}$ in parallel* **do**
2 Build a subgraph $\vec{G}_c \leftarrow \vec{G}[V(c) \cup N_F(c)]$
3 Find all k-cliques except c in $\vec{G}_c$ and add them to C
4 **return** C

k-clique in $C(c)$ shares at least one common node with c. When applying the swap in a clique in $\mathcal{S}$, only its candidate k-cliques have the opportunity to be added to $\mathcal{S}$. Then, we formally propose our swap operation (see Algorithm 5). It utilizes a queue to store cliques in $\mathcal{S}$ that are eligible for swapping. During each iteration, it pops a clique c from the queue and attempts to find a set S_{dis} of disjoint k-cliques using Algorithm 3 among its candidate cliques (Lines 2-4). If it finds a S_{dis} with a size larger than 1, c is removed from $\mathcal{S}$ and the cliques in S_{dis} are added to $\mathcal{S}$ (Lines 5-6), thereby increasing the size of $\mathcal{S}$. Subsequently, the candidate cliques are updated, and any clique c' in $\mathcal{S}$ with $C(c')$ involving new candidate cliques is pushed to the queue for additional swapping (Lines 7-8).

Example 5.3. In Figure 4(b), suppose that $\mathcal{S}$ initially contains two 3-cliques (v_9, v_{10}, v_{11}) and (v_3, v_4, v_5). Suppose that the input queue q contains a clique (v_3, v_4, v_5), we first pop the clique (v_3, v_4, v_5) from q. Its candidate cliques are (v_1, v_2, v_3) and (v_5, v_6, v_7), which is also a set of disjoint k-cliques. Thus, we remove (v_3, v_4, v_5) from $\mathcal{S}$ and add these two candidate cliques to $\mathcal{S}$. After that, the clique (v_9, v_{10}, v_{11}) has a new candidate clique (v_4, v_9, v_{11}), and thus, is pushed to q. Finally, we pop the clique (v_9, v_{10}, v_{11}) from q and immediately return, as it only has one candidate clique, which can not expand $\mathcal{S}$.

Analysis. The swap operation is effective as it can expand $\mathcal{S}$ and lead to achieving a nearly local optimum after several iterations. The time complexity of Algorithm 5 is $O(k \cdot |\mathcal{S}| \cdot \binom{k \cdot cv(G)}{k} \cdot \log \binom{k \cdot cv(G)}{k})$. To explain, for each k-clique, the cost of listing its neighboring k-cliques is $O(\binom{k \cdot cv(G)}{k})$ and it costs $O(\binom{k \cdot cv(G)}{k} \cdot \log \binom{k \cdot cv(G)}{k})$ to find a disjoint k-clique set. Besides, the number of swaps is $O(k \cdot |\mathcal{S}|)$. The space complexity is $O(m + n + |\mathcal{S}| + |C|)$.

5.2 Indexing Structures

Candidate cliques can help accelerate the swap operation. However, listing candidate cliques from scratch for each swap is expensive. Thus, we maintain candidate cliques as our index to reduce the redundant computation cost. Algorithm 6 provides the detail of finding all candidate k-cliques. Since any candidate clique can not neighbor to more than one clique in $\mathcal{S}$, finding all candidate cliques involves identifying candidate cliques of c for each clique c in $\mathcal{S}$ and collecting them to form a set C of candidate cliques. Given a clique c in $\mathcal{S}$, we use $N_F(c)$ to denote the set of free nodes that are the neighbors of any node of c in G. To identify candidate cliques of c, we first build a subgraph $\vec{G}_c$ reduced by nodes in $V(c) \cup N_F(c)$ (Line 2). Subsequently, we find all cliques except c in $\vec{G}_c$ and add them to C (Line 3).

Example 5.4. Figure 4(a) shows an example of identifying candidate cliques when k is 3. $\mathcal{S}$ contains two cliques (v_3, v_4, v_5) and (v_9, v_{10}, v_{11}). For the clique (v_3, v_4, v_5), we first build the subgraph induced by nodes $V(c) \cup N_F(c)$ which is $\{v_3, v_4, v_5, v_1, v_2, v_6, v_7\}$. Then we find one candidate clique (v_1, v_2, v_3) in this subgraph for this clique (v_3, v_4, v_5). Second, for the clique (v_9, v_{10}, v_{11}), as it has no neighboring free nodes, it has no candidate clique.

Algorithm 7: INSERTION($\langle u, v \rangle$)

Input: A directed graph $\vec{G}$, k, $\langle u, v \rangle$
Output: A set $\mathcal{S}$ of disjoint k-cliques
1 **if** *only one node u is a free node* **then**
2 Find new candidate k-cliques containing $\langle u, v \rangle$
3 **if** *found* **then**
4 $c \leftarrow$ the clique in $\mathcal{S}$ containing v
5 Queue $q.push(c)$
6 $TrySwap(q)$
7 **else if** *u and v are both free nodes* **then**
8 **if** *free nodes can form a clique* **then**
9 Add this new clique to $\mathcal{S}$
10 Update the candidate cliques
11 **else**
12 Update the candidate cliques
13 Queue $q \leftarrow \emptyset$
14 Push all cliques c' in $\mathcal{S}$ whose $C(c')$ involves new candidate cliques to q
15 $TrySwap(q)$
16 **return** $\mathcal{S}$

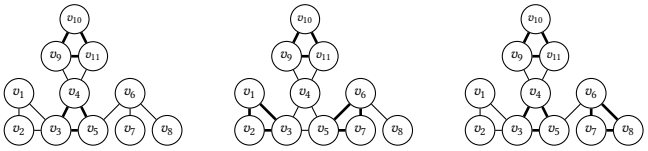
Analysis. The size of candidate k-cliques is bounded by $|\mathcal{S}| \cdot \binom{k \cdot cv(G)}{k}$. The time complexity of Algorithm 6 is $O(|\mathcal{S}| \cdot \binom{k \cdot cv(G)}{k}))$. The space complexity is $O(m+n+|\mathcal{S}|+|\mathcal{C}|)$. In practice, the quantity of candidate cliques is not expected to be large due to the strong constraint of the candidate clique, which requires that internal nodes are either free nodes or belong to the same clique in $\mathcal{S}$.

5.3 Handling Updates

We are now ready to propose our dynamic methods by applying the swap operation and the index. However, the challenges arise when dealing with edge insertions and deletions: determining (i) when to apply the swap operation and (ii) how to identify the set of cliques that have the potential to expand $\mathcal{S}$ as the input of our swap operation. Regarding the second challenge, after performing edge updates, we select the cliques in $\mathcal{S}$ that include new candidate cliques as input for the swap operation. Thus, the answer to the first challenge naturally emerges: the swap operation is applied only when we can identify cliques in $\mathcal{S}$ that involve new candidate cliques. Next, we formally introduce how our insertion and deletion methods handle the above challenges.

5.3.1 Incremental Update. An edge insertion may create new candidate cliques that can be used to expand $\mathcal{S}$ by our swap operation, or it may produce new cliques that can be directly added to $\mathcal{S}$.

Algorithm 7 presents the details of our incremental method that deals with two cases. Suppose that the new edge is $\langle u, v \rangle$. The first case occurs when only one node u is a free node, and the second case occurs when both u and v are free nodes. If neither u nor v are



(a) G_1 (b) $G_2 : G_1 + \langle v_5, v_7 \rangle$ (c) $G_3 : G_1 + \langle v_8, v_7 \rangle$

Figure 4: Insertion example; bold edges form 3-cliques in $\mathcal{S}$

free nodes, nothing needs to be done. In the first case where only u is a free node (Lines 1-6), we first find new candidate k-cliques containing $\langle u, v \rangle$ (Line 2). If new candidate cliques are found, we add the clique containing v in $\mathcal{S}$ to the queue q as the parameter of conducting $TrySwap$ to expand $\mathcal{S}$. When both u and v are free nodes (Lines 7-15), if free nodes can form a new clique among the subgraph reduced by u, v and their neighboring free nodes, we can directly add the new clique to $\mathcal{S}$ and update the corresponding candidate cliques. Notably, we do not conduct $TrySwap$ in this case as the other cliques in $\mathcal{S}$ will not involve new candidate cliques. When free nodes can not produce cliques (Lines 12-15), as this new edge may form new candidate cliques, we first find new candidate cliques. Then, we push any clique c' in $\mathcal{S}$ whose $C(c')$ involves new candidate cliques to q and use $TrySwap$ to expand $\mathcal{S}$.

Example 5.5. Figure 4 shows an example of an edge insertion when k is 3. First, in Figure 4(b), suppose we add an edge $\langle v_5, v_7 \rangle$ from Figure 4(a) where v_7 is a free node. We find a new candidate clique (v_5, v_6, v_7), and thus the clique (v_3, v_4, v_5) is added to the queue. Next, we conduct TrySwap which removes the clique (v_3, v_4, v_5) from $\mathcal{S}$ and adds (v_5, v_6, v_7) and (v_1, v_2, v_3) to $\mathcal{S}$. In Figure 4(c), we add $\langle v_8, v_7 \rangle$ from Figure 4(a) where two nodes are both free nodes. We find a clique (v_6, v_7, v_8) whose nodes are all free nodes. Thus, we add it to $\mathcal{S}$ and return without conducting TrySwap.

Analysis. The bottleneck of the insertion method is the $TrySwap$ procedure. Thus, the time complexity of Algorithm 7 is $O(k \cdot |\mathcal{S}| \cdot \binom{k \cdot cv(G)}{k} \cdot \log \binom{k \cdot cv(G)}{k})$. The space complexity is $O(m+n+|\mathcal{S}|+|C|)$.

5.3.2 Decremental Update. An edge deletion $\langle u, v \rangle$ can lead to the splitting of a clique in $\mathcal{S}$ or candidate cliques containing this edge. Thus, there are two cases to consider. The first case occurs when both u and v are in a clique of $\mathcal{S}$, which results in the splitting of this clique in $\mathcal{S}$ and makes $\mathcal{S}$ not maximal. Thus, we can apply our swap framework in this split clique to expand $\mathcal{S}$. The second case occurs when u and v form candidate k-cliques. We only need to delete such candidate k-cliques.

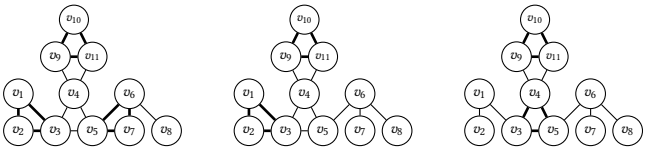
Algorithm 8 presents the details of the edge deletion method. In the case where u and v are in a clique c of $\mathcal{S}$ (Lines 1-4), we push c to the queue and perform $TrySwap$ to expand $\mathcal{S}$. Otherwise, if u and v are not in a clique of $\mathcal{S}$, we directly delete invalid candidate k-cliques containing $\langle u, v \rangle$ (Lines 5-6).

Example 5.6. Figure 5 shows an example of an edge deletion when k is 3. In Figure 5(b), we delete an edge $\langle 5, 7 \rangle$ from Figure 5(a), which splits a clique (v_5, v_6, v_7) in $\mathcal{S}$. As this deleted clique does not have candidate cliques, we can directly return. Then, in Figure 5(c), we delete an edge $\langle 2, 3 \rangle$ from Figure 5(b), which splits a clique (v_1, v_2, v_3) in $\mathcal{S}$. As this deleted clique has a candidate clique (v_3, v_4, v_5), this candidate clique is added to $\mathcal{S}$.

Algorithm 8: DELETION($\langle u, v \rangle$)

Input: A directed graph $\vec{G}$, k, $\langle u, v \rangle$
Output: A set $\mathcal{S}$ of disjoint k-cliques

```
1 if u and v are in a clique of S then
2     Let c be the deleted k-clique in S
3     Queue q.push(c)
4     TrySwap(q)
5 else
6     Delete candidate k-cliques containing ⟨u, v⟩.
7 return S
```



(a) G_1 (b) $G_2 : G_1 - \langle v_5, v_7 \rangle$ (c) $G_3 : G_2 - \langle v_2, v_3 \rangle$

Figure 5: Deletion example; bold edges form 3-cliques in $\mathcal{S}$

Analysis. The bottleneck of the deletion method is also the $TrySwap$ procedure. Thus, the time complexity of Algorithm 8 is $O(k \cdot |\mathcal{S}| \cdot \binom{k \cdot cv(G)}{k} \cdot \log \binom{k \cdot cv(G)}{k})$. The space complexity is $O(m+n+|\mathcal{S}|+|C|)$.

6 EXPERIMENTS

We conduct extensive experiments over 10 real-world graphs on a Linux machine with an Intel Xeon 2.10GHz CPU and 504GB memory. All algorithms are implemented in C++ and compiled using g++ with full optimization. By default, we use 64 threads. The runtime of any algorithm that exceeds 24 hours will be reported as out-of-time "OOT". We also report out-of-memory "OOM" if the memory cost of any algorithm exceeds the limit. We vary k from 3 to 6 as k is usually small in our real application.

6.1 Experimental Setup

Datasets. We test on 10 real graph datasets with different scales, whose statistics are shown in Table 1. All datasets are publicly available on KONECT [33] and Network Repository [41].

Competitors. For static graphs, we have 5 competitors: the exact solution, denoted by OPT, which is calculated by computing the exact MIS solution with [2] in the clique graph; the basic framework on G (Algorithm 2), denoted by HG; the method with a k-clique ordering (Algorithm 3), denoted by GC; the lightweight ordering method without our proposed score-based pruning strategy (Algorithm 4), denoted by L; the lightweight ordering method with our proposed score-based pruning strategy (Algorithm 4), denoted by LP. For dynamic graphs, since there are no existing dynamic algorithms for this problem, we report the update efficiency and compare the size of the updated $\mathcal{S}$ with that of building from scratch.

Implementation details. Notably, although Algorithm 4 and Algorithm 3 can produce the same $\mathcal{S}$ with a fixed total node ordering and a fixed total clique ordering (see Theorem 4.10), we do not maintain the fixed total clique ordering for efficiency considerations. Instead, we only guarantee that when a clique is added to $\mathcal{S}$, its clique score is currently the minimum. In other words, if two

Table 1: Statistics of Datasets. ($K, M, B, T = 10^3, 10^6, 10^9, 10^{12}$)

Name	Dataset	V	E	Number of k-cliques			
				3	4	5	6
FTB	Football	115	613	810	732	473	237
HST	Hamsterster	1.86K	12.5K	16.8K	10K	2.77K	285
FB	Facebook	4K	88K	1.61M	30M	518M	7.83B
FBP	FBPages	28K	206K	393K	837K	2.19M	6.1M
FBW	FBWosn	63.7K	817K	3.5M	13.3M	46.5M	145M
DS	Dogster	260K	2.15M	5.17M	28.5M	131M	475M
SK	Skitter	1.7M	11M	28.8M	149M	1.18B	9.76B
FL	Flickr	1.7M	15.6M	548M	26.7B	1.07T	33.6T
LJ	Livejournal	5.2M	48.7M	311M	11.4B	589B	28.2T
OR	Orkut	3M	117M	628K	3.22B	15.8B	75.2B

cliques have the same clique score, then we directly add the first countered one to $\mathcal{S}$. As a result, the quality of $\mathcal{S}$ between these two algorithms may differ slightly. Our implementation is available at https://github.com/jerchenxin/disjoint-k-clique.

6.2 Evaluation on Static Graphs

In this section, we evaluate the performance of our proposed algorithms in running time, and output quality.

Running time and quality with varying k. Figure 6 shows the running time of each algorithm. To ensure fairness, the running time includes initialization and calculation time. First, OPT is inefficient. OPT runs OOT or OOM even on small datasets since computing the clique graph and the exact MIS is costly. Second, HG is the most efficient, and its running time remains nearly the same with varying k. Third, GC is much slower than LP, and it runs OOM for five datasets when k is large. For example, for FL, LJ, and OR, it runs OOM when k is larger than 3. Fourth, compared with GC, L and LP are efficient and nearly one to two orders of magnitude faster than GC. Compared with LP, when k is 3 and 4, HG is nearly 2X faster in most datasets. As k increases, the running time of L and LP exhibits nearly exponential growth. Finally, compared with L, LP is more and more efficient when k is larger than 3. For example, LP is nearly one order of magnitude faster in LJ when k is 6.

Table 2 presents the quality of $\mathcal{S}$ of each algorithm. Due to the same quality of $\mathcal{S}$ of L and LP, we only report the quality of $\mathcal{S}$ of LP. In this table, we report the exact size of $\mathcal{S}$ for OPT and HG, and the relative size for GC and LP, denoted by Δ, which is the difference between the size of their $\mathcal{S}$ and that of HG. For example, when k is 3 in FB, the size of $\mathcal{S}$ of HG is 1195 while that of GC and LP is 1235. First, we can see that LP and OPT can produce nearly the same $\mathcal{S}$, which indicates that LP can produce a high-quality solution. Second, we can see that the size of $\mathcal{S}$ of GC and LP is nearly the same. The slight difference is due to the absence of a strict total node and clique ordering, which is done for efficiency considerations. Third, compared with HG, the size of $\mathcal{S}$ of LP is nearly up to 13.3% larger. For example, in OR, when k is 6, the size of $\mathcal{S}$ of LP is 13.3% larger. In LJ, when k is 6, the size of $\mathcal{S}$ LP is 11.7% larger. Thus, we conclude that LP can produce a near-optimal $\mathcal{S}$.

The time breakdown of each step of LP. In this experiment, we demonstrate the efficiency of LP by presenting the time breakdown of each step. There are three costly steps in LP (see Algorithm 4): the node score calculation (Lines 1-4), the heap initialization (Line 6), and the calculation of $\mathcal{S}$ (Line 7). Table 3 shows the time breakdown of each step where S1, S2, and S3 correspond to these three steps, respectively. We report the ratio of the cost of each step. We can observe that S1 accounts for most of the running time. For example, in FL, when k is 6, S1 costs 99.73%, indicating the efficiency of our proposed score-driven pruning strategy. For some cases, like FB when k is 4, S3 will account for most of the running time. To explain, LP involves redundant computation in S3. When the clique graph is extremely dense, the cost of S3 will increase. However, it is noteworthy that even when S3 accounts for most of the running time in certain cases, the cost of S1 can still not be omitted, indicating that redundant computation is not the major problem in most cases.

Table 2: Size of $\mathcal{S}$. (For GC and LP, Δ is the difference between the size of their $\mathcal{S}$ with that of HG)

Name	k=3				k=4				k=5				k=6			
	OPT	HG	GC (Δ)	LP (Δ)	OPT	HG	GC (Δ)	LP (Δ)	OPT	HG	GC (Δ)	LP (Δ)	OPT	HG	GC (Δ)	LP (Δ)
FTB	OOT	32	4	4	25	24	-1	-1	16	16	0	0	11	11	0	0
HST	OOT	201	10	10	OOT	52	6	6	15	13	1	1	5	4	1	1
FB	OOT	1,195	40	40	OOM	784	48	48	OOM	561	37	37	OOM	413	OOM	31
FBP	OOT	5,732	357	348	OOT	2,888	254	249	OOM	1,602	163	164	OOM	967	88	106
FBW	OOT	13,114	1,112	1,121	OOM	7,041	661	660	OOM	4,146	443	447	OOM	2,606	307	309
DS	OOM	19,974	364	357	OOM	5,044	231	234	OOM	1,787	127	127	OOM	797	78	78
SK	OOM	132,009	4,458	4,495	OOM	31,775	2,054	2,058	OOM	10,320	985	987	OOM	4,354	462	480
FL	OOM	132,308	8,856	8,830	OOM	51,615	OOM	4,069	OOM	24,220	OOM	1,556	OOM	12,937	OOM	307
LJ	OOM	874,457	74,976	75,026	OOM	430,014	OOM	41,239	OOM	232,180	OOM	24,684	OOM	133,795	OOM	14,877
OR	OOM	861,315	54,590	54,556	OOM	513,758	OOM	49,093	OOM	323,078	OOM	38,041	OOM	212,440	OOM	28,186

Table 3: The time breakdown of each step of LP. (S1, S2, and S3 correspond to the node score calculation, the heap initialization, and the calculation of $\mathcal{S}$, respectively). The ratio of the cost of each step is reported.

Name	$k=3$			$k=4$			$k=5$			$k=6$		
	S1	S2	S3	S1	S2	S3	S1	S2	S3	S1	S2	S3
FTB	**90.66%**	6.35%	2.99%	**92.97%**	5.04%	1.99%	**94.88%**	3.88%	1.24%	**94.47%**	3.50%	2.02%
HST	**87.5%**	10.00%	2.50%	**84.21%**	10.53%	5.26%	**88.89%**	5.56%	5.56%	**90.00%**	5.00%	5.00%
FB	**67.89%**	8.26%	23.85%	33.05%	2.12%	**64.83%**	15.41%	0.45%	**84.13%**	28.27%	0.18%	**71.55%**
FBP	**77.22%**	15.00%	7.78%	**80.21%**	8.56%	11.23%	**76.85%**	7.88%	15.27%	**73.66%**	4.39%	21.95%
FBW	**72.46%**	13.56%	13.98%	**67.27%**	8.78%	23.95%	**59.53%**	4.68%	35.79%	**51.87%**	3.45%	44.68%
DS	**85.88%**	8.70%	5.42%	**83.78%**	7.83%	8.39%	**79.76%**	6.62%	13.61%	**82.79%**	4.09%	13.12%
SK	**73.34%**	19.44%	7.22%	**73.45%**	17.97%	8.58%	**69.93%**	16.37%	13.70%	**70.62%**	10.85%	18.53%
FL	**71.57%**	16.81%	11.62%	**70.49%**	6.37%	23.14%	**94.80%**	0.52%	4.69%	**99.73%**	0.09%	0.18%
LJ	**66.99%**	12.76%	20.25%	34.55%	4.19%	**61.26%**	24.96%	1.06%	**73.98%**	**89.72%**	0.15%	10.13%
OR	**57.00%**	7.01%	35.99%	**49.06%**	5.53%	45.40%	41.74%	4.57%	**53.69%**	40.42%	3.24%	**56.34%**

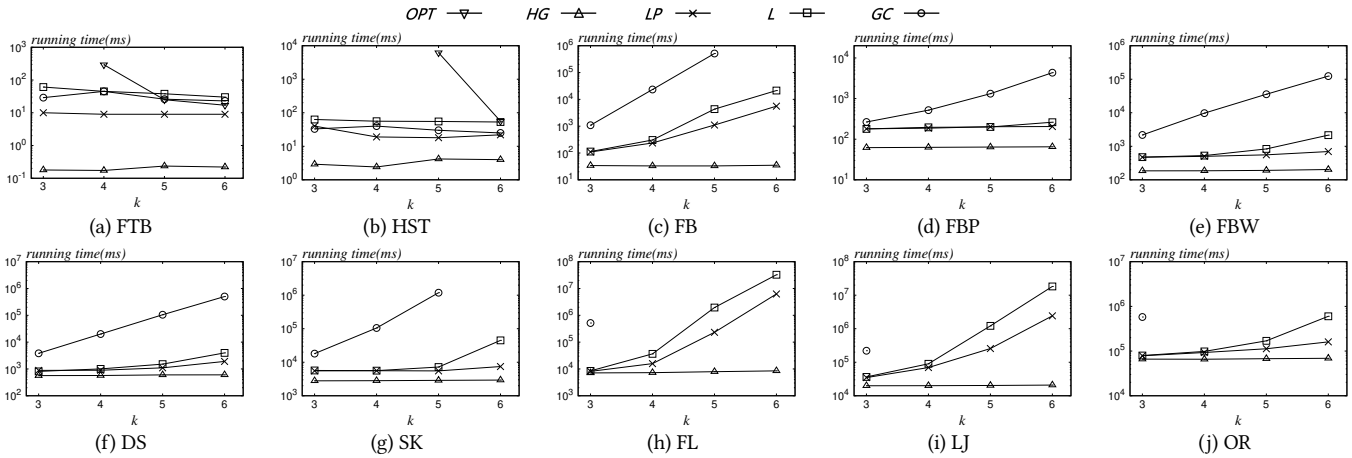
Table 4: Running time (s) on synthetic datasets.

Degree	k=3			k=4			k=5			k=6		
	HG	GC	LP	HG	GC	LP	HG	GC	LP	HG	GC	LP
8	1.2	3.28	3.98	1.22	2.29	3.24	1.19	1.9	2.77	1.14	1.49	1.96
16	2.2	9.36	6.98	2.41	9.96	6.14	2.3	6.26	5.34	2.4	4.1	4.3
32	4.15	35.5	14.3	4.29	116	15.4	4.45	146	15.3	4.64	123	13.6
64	9.85	194	32.2	8.78	1.66K	47.3	8.86	4.83K	82.4	9.14	OOM	133

Table 5: Size of $\mathcal{S}$ on synthetic datasets.

Degree	k=3			k=4			k=5			k=6		
	HG	GC (Δ)	LP (Δ)	HG	GC (Δ)	LP (Δ)	HG	GC (Δ)	LP (Δ)	HG	GC (Δ)	LP (Δ)
8	275,636	19,394	19,164	160,284	14,764	14,693	57,106	717	719	0	0	0
16	303,408	7,277	7,291	206,693	17,764	17,695	138,749	24,804	24,783	88,509	12,026	12,050
32	317,417	3,509	3,444	228,178	7,460	7,446	171,736	12,707	12,710	130,654	18,749	18,735
64	324,964	1,629	1,683	238,631	3,278	3,273	185,736	5,421	5,405	149,210	OOM	8,131

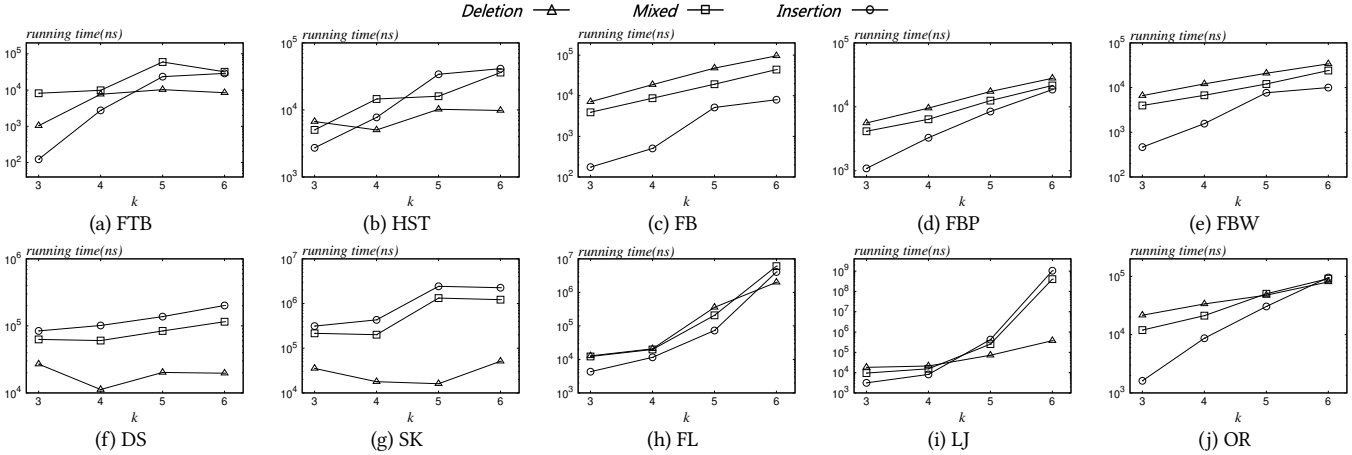
Performance on synthetic datasets. In this experiment, we vary the graph density to evaluate the scalability of HG, GC, and LP. We generate several random graphs under the Watts-Strogatz model [44] with the number of nodes equal to $n = 1M$, while the average node degree is varied from 8 to 64, leading to the number of edges varied from $4M$ to $32M$ respectively. Table 4 presents the running time, and Table 5 presents the size of $\mathcal{S}$. We can first observe that both the size of $\mathcal{S}$ and the running time of each method increase as the graph becomes denser. Besides, the running time of HG remains nearly the same as k increases. The running time of GC and LP depends on the number of k-cliques as collecting all k-cliques and the node score calculation usually consume most of the time. For example, when the average degree is 32, the running time of LP first increases and then decreases when $k \geq 4$.



(a) FTB (b) HST (c) FB (d) FBP (e) FBW

(f) DS (g) SK (h) FL (i) LJ (j) OR

Figure 6: Average running time in milliseconds with varying k.



(a) FTB (b) HST (c) FB (d) FBP (e) FBW

(f) DS (g) SK (h) FL (i) LJ (j) OR

Figure 7: Average update time in nanosecond seconds with varying k in deletion, insertion, and the mixed workload.

6.3 Evaluation on Dynamic Graphs

In this section, we evaluate the performance of our proposed dynamic algorithms in indexing time, index size, update performance, and the quality of $\mathcal{S}$ after updates.

Index construction. First, we evaluate the indexing time and the index size of the proposed index. Table 6 shows the indexing time and the index size, with the number of candidate cliques representing the index size. First, we can observe that indexing time increases with the increase of the index size. For example, in FBP, with the increase of k, the index size increases, leading to an increase in indexing time. Second, we can observe our index is efficient to build, and has a small size. For instance, in OR when k is 6, it only has 1.92M candidate cliques while it has 75.2B 6-cliques. To explain, nodes in a candidate clique can be either free nodes or belong to the same clique in $\mathcal{S}$. Such a strict constraint limits the index size.

Update performance under three workloads. Next, we evaluate the update performance under three workloads: 10K edge deletions, 10K edge insertions, and mixed workloads with 20K updates. For FTB and HST, we use 10 and 1K edge deletions and insertions, and mixed workloads with 20 and 2K updates, respectively, since the graph is small. For the first and second workloads, we first randomly and uniformly select 10K edges. Then, we delete these edges to report the performance under edge deletions and add them back to report the performance under edge insertions. For the mixed

Table 6: Indexing time and index size.

Dataset	Indexing Time (ms)				Index Size			
	3	4	5	6	3	4	5	6
FTB	7.1	11.1	11.1	11.3	86	149	419	226
HST	9.91	17.7	11.6	15.5	1.01K	327	274	20
FB	10.5	9.57	19.6	43.4	1.45K	3.03K	3.87K	16.8K
FBP	37.3	41.7	44.9	61.7	9.61K	10.9K	16.3K	25.7K
FBW	58.2	63.7	73.1	88.6	5.98K	9.41K	16.6K	30.2K
DS	307	207	192	190	109K	15.3K	5.18K	3.36K
SK	2.73K	1.83K	1.71K	2.06K	1.51M	453K	188K	132K
FL	1.46K	1.52K	1.84K	2.66K	173K	103K	274K	419K
LJ	5.59K	5.8K	7.16K	14.6K	983K	1.13M	3M	4.73M
OR	4.97K	4.95K	5.19K	7.36K	278K	612K	1.07M	1.92M

Table 7: Quality of $\mathcal{S}$ after updates. (Δ is the difference of the size of $\mathcal{S}$ compared with that of building from scratch)

Dataset	After Deletion (Δ)				After Insertion (Δ)				After Mixed Updates (Δ)			
	3	4	5	6	3	4	5	6	3	4	5	6
FTB	0	0	0	-1	0	0	0	0	0	0	0	0
HST	0	3	-1	0	0	2	1	0	-1	-1	0	0
FB	-13	-9	-14	-12	-23	-28	-36	-32	-35	-30	-28	-20
FBP	-18	-24	-2	6	-7	-12	-14	-1	-58	-32	-17	-16
FBW	-110	-63	-49	-11	-45	-36	-29	-2	-153	-120	-89	-61
DS	38	1	2	1	59	11	3	0	55	-2	-8	-11
SK	35	-3	6	-4	33	26	10	8	-3	20	-5	0
FL	-16	-30	193	367	14	0	198	378	-34	-60	190	385
LJ	42	68	51	68	96	88	70	132	-80	21	-17	64
OR	-78	-43	36	29	28	-4	34	37	-128	-94	-41	-38

workloads, we generate 20K updates including 10K insertions and 10K deletions. All edges are also selected randomly and uniformly. For 10K insertions, we will first delete them from G to form a new graph G'. Then, we evaluate the performance in the mixed workload by applying 20K updates in G'. For each workload, we report the average running time in Figure 7 and the size of $\mathcal{S}$ after updates in Table 7.

Figure 7 presents the average update time. First, we can observe that the average update time of three workloads increases with the increase of k as a larger k incurs more subgraph computation costs. However, For DS and SK, the average deletion time drops when k is 4 and 5. To explain, when k is 4 and 5, the size of candidate cliques of DS and SK significantly decreases, which leads to the decreased cost of maintaining the candidate cliques. In addition, compared with building from scratch, our proposed insertion and deletion algorithms are efficient. For example, for OR, when k is 3, the time of building from scratch equals that of 3.6M edge deletion operations, that of 48.6M edge deletion operations, and that of 6.5M mixed update operations. It shows that only with such a large amount of updates, the total update time is the same as that of building from scratch, which indicates the efficiency of our dynamic algorithms.

Besides, Table 7 shows the size of $\mathcal{S}$ after updates compared with that of building from scratch. We can observe that $\mathcal{S}$ preserves a high quality. For example, for OR after deletion when k is 3, $\mathcal{S}$ decreases by 78, which is only 0.08% of the size of $\mathcal{S}$. For datasets like LJ, the size of $\mathcal{S}$ even increases. To explain, our swap operation can reach a nearly local optimum, and thus, its size may be larger than that of building from scratch. Thus, we can conclude that our proposed dynamic methods are efficient and effective.

7 CONCLUSIONS

In this paper, we study a new problem, *Maximum Disjoint k-Clique Set*, which finds essential real-world applications in online games but is proved to be NP-hard. To address that, we propose an efficient lightweight method that achieves a k-approximation to the optimal. Specifically, we develop several optimization techniques, including the node ordering method, the k-clique ordering method, and a lightweight implementation. Besides, to handle dynamic graphs, we devise an efficient indexing method with careful swapping operations, leading to the successful maintenance of a near-optimal result with frequent updates in the graph. In various experiments on several large graphs with dynamic settings, our proposed approaches outperform the competitors siginicantly in terms of both efficiency and effectiveness.

REFERENCES

[1] Esra Akbas and Peixiang Zhao. 2017. Truss-based Community Search: a Truss-equivalence Based Indexing Approach. *Proc. VLDB Endow.* 10, 11 (2017), 1298–1309.

[2] Takuya Akiba and Yoichi Iwata. 2016. Branch-and-reduce exponential/FPT algorithms in practice: A case study of vertex cover. *Theor. Comput. Sci.* 609 (2016), 211–225.

[3] Gary D. Bader and Christopher W. V. Hogue. 2003. An automated method for finding molecular complexes in large protein interaction networks. *BMC Bioinform.* 4 (2003), 2.

[4] Piotr Berman. 2000. A $d/2$ Approximation for Maximum Weight Independent Set in d-Claw Free Graphs. In *Algorithm Theory - SWAT 2000, 7th Scandinavian Workshop on Algorithm Theory, Bergen, Norway, July 5-7, 2000, Proceedings (Lecture Notes in Computer Science)*, Vol. 1851. Springer, 214–219.

[5] Piotr Berman and Toshihiro Fujito. 1999. On Approximation Properties of the Independent Set Problem for Low Degree Graphs. *Theory Comput. Syst.* 32, 2 (1999), 115–132.

[6] Piotr Berman and Piotr Krysta. 2003. Optimizing misdirection. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA*. ACM/SIAM, 192–201.

[7] Barun Chandra and Magnús M. Halldórsson. 2001. Greedy Local Improvement and Weighted Set Packing Approximation. *J. Algorithms* 39, 2 (2001), 223–240.

[8] Lijun Chang, Wei Li, and Wenjie Zhang. 2017. Computing A Near-Maximum Independent Set in Linear Time by Reducing-Peeling. In *SIGMOD*. ACM, 1181–1196.

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

[10] Marek Cygan. 2013. Improved Approximation for 3-Dimensional Matching via Bounded Pathwidth Local Search. In *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, 26-29 October, 2013, Berkeley, CA, USA*. IEEE Computer Society, 509–518.

[11] Maximilien Danisch, Oana Balalau, and Mauro Sozio. 2018. Listing k-cliques in Sparse Real-World Graphs. In *WWW*. ACM, 589–598.

[12] Dongsheng Duan, Yuhua Li, Ruixuan Li, and Zhengding Lu. 2012. Incremental K-clique clustering in dynamic social networks. *Artif. Intell. Rev.* 38, 2 (2012), 129–147.

[13] Fanny Dufossé, Kamer Kaya, Ioannis Panagiotas, and Bora Uçar. 2019. Effective Heuristics for Matchings in Hypergraphs. In *Analysis of Experimental Algorithms - Special Event, SEA^2 2019, Kalamata, Greece, June 24-29, 2019, Revised Selected Papers (Lecture Notes in Computer Science)*, Vol. 11544. Springer, 248–264.

[14] Jack Edmonds. 1965. Paths, trees, and flowers. *Canadian Journal of mathematics* 17 (1965), 449–467.

[15] Shimon Even and Oded Kariv. 1975. An O(n^2.5) Algorithm for Maximum Matching in General Graphs. In *16th Annual Symposium on Foundations of Computer Science, Berkeley, California, USA, October 13-15, 1975*. IEEE Computer Society, 100–112.

[16] Manuela Fischer, Mohsen Ghaffari, and Fabian Kuhn. 2017. Deterministic Distributed Edge-Coloring via Hypergraph Maximal Matching. In *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017, Berkeley, CA, USA, October 15-17, 2017*. IEEE Computer Society, 180–191.

[17] Fedor V. Fomin, Fabrizio Grandoni, and Dieter Kratsch. 2009. A measure & conquer approach for the analysis of exact algorithms. *J. ACM* 56, 5 (2009), 25:1–25:32.

[18] Martin Fürer and Huiwen Yu. 2013. Approximate the k-Set Packing Problem by Local Improvements. *CoRR* abs/1307.2262 (2013).

[19] Harold N. Gabow. 1976. An Efficient Implementation of Edmonds' Algorithm for Maximum Matching on Graphs. *J. ACM* 23, 2 (1976), 221–234.

[20] Enrico Gregori, Luciano Lenzini, and Simone Mainardi. 2013. Parallel (k)-Clique Community Detection on Large-Scale Networks. *IEEE Trans. Parallel Distributed Syst.* 24, 8 (2013), 1651–1660.

[21] Magnús M. Halldórsson. 1995. Approximating Discrete Collections via Local Improvements. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, 22-24 January 1995. San Francisco, California, USA*. ACM/SIAM, 160–169.

[22] Magnús M. Halldórsson and Jaikumar Radhakrishnan. 1997. Greed is Good: Approximating Independent Sets in Sparse and Bounded-Degree Graphs. *Algorithmica* 18, 1 (1997), 145–163.

[23] Oussama Hanguir and Clifford Stein. 2020. Distributed Algorithms for Matching in Hypergraphs. In *WAOA (Lecture Notes in Computer Science)*, Vol. 12806. Springer, 30–46.

[24] Fei Hao, Geyong Min, Zheng Pei, Doo-Soon Park, and Laurence T. Yang. 2017. K-Clique Community Detection in Social Networks Based on Formal Concept Analysis. *IEEE Syst. J.* 11, 1 (2017), 250–259.

[25] David G. Harris. 2018. Distributed approximation algorithms for maximum matching in graphs and hypergraphs. *CoRR* abs/1807.07645 (2018).

[26] Takanori Hayashi, Takuya Akiba, and Ken-ichi Kawarabayashi. 2016. Fully Dynamic Shortest-Path Distance Query Acceleration on Massive Networks. In *CIKM*. ACM, 1533–1542.

[27] Hao Huang, Po-Shen Loh, and Benny Sudakov. 2012. The Size of a Hypergraph and its Matching Number. *Comb. Probab. Comput.* 21, 3 (2012), 442–450.

[28] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying k-truss community in large and dynamic graphs. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*. ACM, 1311–1322.

[29] Cor A. J. Hurkens and Alexander Schrijver. 1989. On the Size of Systems of Sets Every t of Which Have an SDR, with an Application to the Worst-Case Ratio of Heuristics for Packing Problems. *SIAM J. Discret. Math.* 2, 1 (1989), 68–72.

[30] Guangda Huzhang, Xin Huang, Shengyu Zhang, and Xiaohui Bei. 2017. Online Roommate Allocation Problem. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*. ijcai.org, 235–241.

[31] Tiko Kameda and J. Ian Munro. 1974. A O(|V|*|E|) algorithm for maximum matching of graphs. *Computing* 12, 1 (1974), 91–98.

[32] Richard M. Karp. 1972. Reducibility Among Combinatorial Problems. In *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA (The IBM Research Symposia Series)*. Plenum Press, New York, 85–103.

[33] Jérôme Kunegis. 2013. KONECT: the Koblenz network collection. In *WWW*. 1343–1350.

[34] Matthieu Latapy. 2008. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical computer science* 407, 1-3 (2008), 458–473.

[35] Yu Liu, Jiaheng Lu, Hua Yang, Xiaokui Xiao, and Zhewei Wei. 2015. Towards Maximum Independent Sets on Massive Graphs. *Proc. VLDB Endow.* 8, 13 (2015), 2122–2133.

[36] Silvio Micali and Vijay V. Vazirani. 1980. An O(sqrt(|v|) |E|) Algorithm for Finding Maximum Matching in General Graphs. In *21st Annual Symposium on Foundations of Computer Science, Syracuse, New York, USA, 13-15 October 1980*. IEEE Computer Society, 17–27.

[37] Gergely Palla, Albert-László Barabási, and Tamás Vicsek. 2007. Quantifying social group evolution. *Nature* 446, 7136 (2007), 664–667.

[38] Yue Pang, Lei Zou, and Yu Liu. 2023. IFCA: Index-Free Community-Aware Reachability Processing Over Large Dynamic Graphs. In *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*. IEEE, 2220–2234.

[39] Christos H Papadimitriou. 2003. Computational complexity. In *Encyclopedia of computer science*. 260–265.

[40] Chengzhi Piao, Weiguo Zheng, Yu Rong, and Hong Cheng. 2020. Maximizing the Reduction Ability for Near-maximum Independent Set Computation. *Proc. VLDB Endow.* 13, 11 (2020), 2466–2478.

[41] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*. http://networkrepository.com

[42] Maxim Sviridenko and Justin Ward. 2013. Large Neighborhood Local Search for the Maximum Set Packing Problem. In *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part I (Lecture Notes in Computer Science)*, Vol. 7965. Springer, 792–803.

[43] Charalampos E. Tsourakakis. 2015. The K-clique Densest Subgraph Problem. In *WWW*. ACM, 1122–1132.

[44] Duncan J Watts and Steven H Strogatz. 1998. Collective dynamics of 'small-world'networks. *nature* 393, 6684 (1998), 440–442.

[45] Hao Wei, Jeffrey Xu Yu, Can Lu, and Ruoming Jin. 2018. Reachability querying: an independent permutation labeling approach. *VLDB J.* 27, 1 (2018), 1–26.

[46] Mingyu Xiao and Hiroshi Nagamochi. 2017. Exact algorithms for maximum independent set. *Inf. Comput.* 255 (2017), 126–146.

[47] Tianyang Xu, Zhao Lu, and Yuanyuan Zhu. 2022. Efficient Triangle-Connected Truss Community Search In Dynamic Graphs. *Proc. VLDB Endow.* 16, 3 (2022), 519–531.

[48] Michael Yu, Lu Qin, Ying Zhang, Wenjie Zhang, and Xuemin Lin. 2020. Aot: Pushing the efficiency boundary of main-memory triangle listing. In *DASFAA*. 516–533.

[49] Zhirong Yuan, You Peng, Peng Cheng, Li Han, Xuemin Lin, Lei Chen, and Wenjie Zhang. 2022. Efficient k-clique Listing with Set Intersection Speedup.. In *ICDE*. 1955–1968.

[50] Mengxuan Zhang, Lei Li, Wen Hua, Rui Mao, Pingfu Chao, and Xiaofang Zhou. 2021. Dynamic Hub Labeling for Road Networks. In *ICDE*. IEEE, 336–347.

[51] Mengxuan Zhang, Lei Li, Wen Hua, and Xiaofang Zhou. 2021. Efficient 2-Hop Labeling Maintenance in Dynamic Small-World Networks. In *ICDE*. IEEE, 133–144.

[52] Andy Diwen Zhu, Wenqing Lin, Sibo Wang, and Xiaokui Xiao. 2014. Reachability queries on large dynamic graphs: a total order approach. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*. ACM, 1323–1334.