Scaling Up k-Clique Percolation Community Detection

Anonymous Author(s)

Abstract

Overlapping communities are pervasive in real-world networks, where vertices often participate in multiple communities simultaneously. The k-clique percolation community (KCPC) model represents a fundamental paradigm for mining overlapping communities. However, existing KCPC mining methods are often hampered by inefficiency and scalability challenges, hindering their applicability to large-scale networks. To address these challenges, we propose several novel and efficient approaches for KCPC mining from perspectives of maximal cliques and k-cliques. Specifically, we first present a novel concept, termed Quasi-KCPC, which represents an incomplete KCPC and can be efficiently obtained as a byproduct during the maximal clique enumeration procedure. Based on Quasi-KCPC, we first propose a maximal clique enumeration-based solution that builds upon existing maximal clique adjacency graph traversal methods, but achieves improved efficiency by using Quasi-KCPC to dramatically reduce the scale of the maximal clique adjacency graph. Additionally, we propose a novel k-clique listing-based solution, which adopts a different strategy: it first enumerates $(k-1)$-cliques and then connects the k-cliques sharing these $(k-1)$-cliques into KCPC. Our method further improves efficiency by shifting the connection target from k-cliques to maximal cliques and employing Quasi-KCPC to significantly prune the k-clique enumeration tree. We also propose dynamic update algorithms for KCPC to handle dynamic addition and deletion of vertices and edges, enabling real-time analysis of KCPC. Extensive experiments on 12 large real-world graphs demonstrate the superiority of our algorithms, which can be up to two orders of magnitude faster than existing state-of-the-art solutions in KCPC mining, and almost two orders of magnitude faster than recomputation strategy in dynamic KCPC updates.

1 Introduction

Community detection is a well-studied problem, with applications in diverse domains like social network analysis [12, 54] and biological networks mining [20]. Real-world networks often exhibit vertices that participate in multiple communities [2, 47, 57], leading to a special focus on overlapping community detection (for a comprehensive overview, see [24, 57]). The most fundamental concept for detecting such overlapping communities is the k-Clique Percolation Community (KCPC) [47]. Specifically, KCPC defines a community as a connected and maximal set of k-cliques. Given an integer k, two cliques are considered adjacent if they share $k-1$ vertices. For a given graph G, each k-clique is treated as a vertex, and their adjacency relationships form a graph, which we refer to as the *k-clique graph* G_C. A KCPC in G corresponds to a connected component of G_C.

The KCPC model has been widely used in analyzing social networks [1, 30, 46, 53], Internet topology [23], keyword networks [26], and extended to fields such as medicine [31, 35, 56], biology [25, 42, 59], psychology [50, 52], and statistical physics [11, 16, 37, 38]. It is widely regarded as an effective tool for overlapping community detection. For example, KCPC has been applied to model overlapping communities in Twitter data [53] and human mobility networks [30], and to study the impact of centrality measures on community structures [1]. In medicine, it has helped identify gene and protein communities linked to diseases such as Alzheimer's [56] and cancer [31]. Applications also include plant gene network analysis [25, 42, 59], psychological symptom networks [50, 52], and research on network percolation theory in statistical physics [11, 16, 37, 38].

Although the KCPC model has been successfully applied in various domains, the existing KCPC mining algorithms face significant limitations when dealing with large-scale networks due to their high computational time overheads. These algorithms can be classified into two main types. The first type is based on maximal cliques [47, 51]. These algorithms typically operate in two steps: first, they construct a maximal clique adjacency graph G_ϕ, where each vertex represents a maximal clique, and two maximal cliques are connected by a weighted edge whose weight corresponds to the number of common vertices shared between the two cliques. Subsequently, the KCPC is derived by identifying the connected components in G_ϕ with edges whose weights are no less than $k-1$. This approach is based on the principle that for a clique of size greater than k, all of its k-clique subgraphs belong to the same KCPC. However, the main challenge with this maximal cliques-based approach lies in the fact that the number of edges in G_ϕ for large-scale networks can be prohibitively large (often reaching magnitudes of 10^{13} or more, as shown in Table 2), making this approach infeasible for real-world applications.

The second type of algorithm is based on k-clique listing [5, 33]. The key idea is to first enumerate all $(k-1)$-cliques and then connect all k-cliques that share each $(k-1)$-clique. However, the primary limitation of this approach is its requirement to exhaustively list all k-cliques. Given that the number of k-cliques typically grows exponentially as k increases [32, 39, 58], this approach becomes computationally infeasible for a relatively large k value (e.g., $k \geq 7$).

To improve the efficiency of KCPC mining, we propose a novel maximal clique enumeration-based solution, QKC, and two novel k-clique listing-based solutions, KCL-MCL and KCL-QKC. Among them, QKC introduces a key innovation: reducing the size of the maximal clique adjacency graph (MCAG) through a novel concept called Quasi-KCPC. A Quasi-KCPC groups maximal cliques that belong to the same KCPC, serving as an intermediate and compact approximation of the final KCPCs. QKC adopts a two-stage approach. In the first stage, it exploits the structure of the maximal clique enumeration tree to generate Quasi-KCPCs during enumeration. Each maximal clique is assigned to a Quasi-KCPC. In the second stage, Quasi-KCPCs are treated as super vertices to build a much smaller adjacency graph, dramatically reducing traversal costs for mining the final KCPCs. This design addresses the key bottleneck in traditional maximal clique enumeration–based methods, where the MCAG size becomes prohibitive. Our experiments show that the number of Quasi-KCPCs is typically an order of magnitude smaller than the number of maximal cliques, reducing about 90% of the vertices in the MCAG. As a result, QKC achieves higher efficiency compared to existing maximal clique enumeration–based solutions such as [47, 51].

Unlike traditional k-clique listing methods [5, 33] that connect k-cliques via shared $(k-1)$-cliques, KCL-MCL and KCL-QKC connect maximal cliques sharing a $(k-1)$-clique. This shift from k-cliques to maximal cliques enables significant pruning of the k-clique search space. By first enumerating all maximal cliques, we exploit the key property that if maximal cliques sharing a k_1-clique

($k_1 < k-1$) are already in the same KCPC, then any $(k-1)$-clique containing C_{k_1} can not merge different KCPC components. Thus, large portions of the search tree can be pruned early. This pruning strategy can not be applied to traditional k-clique listing based method, because k-cliques can not be listed in advance due to the exponentially growing number of k-cliques with growing k [32, 39, 58]. Compared to k-cliques, enumerating maximal cliques can achieve higher efficiency in real-world datasets [14], and the number of maximal cliques and the search space are not affected by k. Another key feature is that we can further enhance the pruning effect using Quasi-KCPC. This is because the maximal cliques within a Quasi-KCPC are already confirmed to belong to the same KCPC component, allowing pruning conditions to be met more quickly. Our experiments show that utilizing Quasi-KCPCs can reduce the number of enumeration tree nodes by nearly two orders of magnitude, i.e., prune 99% of the search space. Thus, our best algorithm KCL-QKC can be up to two orders of magnitude faster than the state-of-the-art methods.

To support dynamic KCPC updates in dynamic graphs, we propose efficient update algorithms for vertex and edge additions and deletions. These algorithms adopt local update strategies and integrate the maintenance of maximal cliques and KCPC through a single framework. Experiments show that our update algorithms are nearly two orders of magnitude faster than recomputation strategy.

To summarize, our main contributions are as follows.

Novel maximal clique enumeration-based solution. We propose a novel two-stage KCPC mining algorithm based on maximal clique enumeration (QKC). The key novelty of this solution lies in introducing a new concept: Quasi-KCPC, and utilizing it to significantly reduce the size of the maximal clique adjacency graph, thereby substantially improving efficiency. Empirical results indicate that 90% of vertices in the maximal clique adjacency graph can be reduced by using our solution.

Novel k-clique listing-based methods. We present two novel k-clique listing-based algorithm, namely KCL-MCL and KCL-QKC respectively. The key novelty of these approaches lies in leveraging maximal cliques or Quasi-KCPCs to significantly prune the search space during the k-clique listing process. This allows the algorithms to avoid exhaustively enumerating all k-cliques, thereby substantially improving efficiency. Empirical results suggest that our approach can prune up to 99% of the k-clique enumeration space, leading to a dramatic reduction in computational overhead.

Efficient KCPC dynamic maintenance. We propose efficient dynamic update algorithms for KCPC under edge and vertex additions and deletions, achieving nearly two orders of magnitude speedup over recomputation by leveraging local updates and integrating maximal clique and KCPC maintenance.

Extensive experiments. We conduct extensive experiments on 12 large real-world graphs, and the results demonstrate that the proposed methods significantly outperform all existing state-of-the-art algorithms. Notably, our best algorithm, KCL-QKC, can be up to two orders of magnitude faster than all existing methods. For example, on dataset Wikitop (1, 791, 488 vertices, 25, 444, 207 edges, $k = 9$), KCL-QKC takes only 401 seconds to compute all KCPCs while the best existing algorithm consumes 50, 399 seconds.

Reproducibility. We release the source code at https://anonymous.4open.science/r/kcpc-efj.

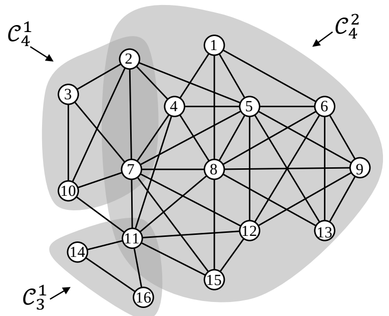


Figure 1: An example graph G and overlapping communities

2 Preliminaries

Let $G = (V, E)$ be an undirected graph, where V and E are sets of vertices and edges respectively. For a vertex $u \in V$, $N_G(u) = \{v|(u,v) \in E\}$ is the set of neighbors of u in G, $d_G(u) = |N_G(u)|$ is the degree of u. A graph $G_S = (V_S, E_S)$ is a subgraph of $G = (V, E)$ if $V_S \subseteq V$ and $E_S \subseteq E$. A subgraph $G_S = (V_S, E_S)$ of G is induced by V_S if $E_S = \{(u,v)|u, v \in V_S, (u,v) \in E\}$.

Definition 1 (Degeneracy). *The degeneracy of a graph G is the minimum integer δ that each subgraph G_S of G contains a vertex u with $d_{G_S}(u) \le \delta$.*

Definition 2 (Degeneracy ordering). *Given a graph $G = (V, E)$, a permutation $(v_1, v_2, ..., v_n)$, $n = |V|$ of vertices in V is a degeneracy ordering of G if for each $i = 1, 2, ..., n$, v_i has the minimum degree in subgraph of G induced by $\{v_i, ..., v_n\}$.*

Degeneracy is a well-known metric to measure the sparsity of a graph [40]. Both the degeneracy and degeneracy ordering of a graph can be computed in linear time [4].

Definition 3 (Clique). *Given a graph $G = (V, E)$, C is a clique of G if C is a set of vertices in G and the induced subgraph by C is a complete graph, i.e., $\forall u, v \in C, (u,v) \in E$.*

A k-clique C is a clique with k vertices, denoted by C_k. A clique C is a maximal clique if there is no other clique C' that $C \subset C'$.

Definition 4 (k-clique Adjacency). *Given two k-cliques C_k^1, C_k^2 of graph G, C_k^1 and C_k^2 are adjacent if they share $k-1$ vertices, i.e., $|C_k^1 \cap C_k^2| = k - 1$.*

Definition 5 (k-clique Connectivity). *Given two k-cliques C_k^a and C_k^b of graph G, if there is a sequence of k-cliques $C_k^a = C_k^1, C_k^2, ..., C_k^n = C_k^b$ that $\forall i = 1, ..., n-1$, C_k^i and C_k^{i+1} are adjacent, then C_k^a and C_k^b are connected.*

Based on Definition 4 and Definition 5, we define KCPC as follows.

Definition 6 (k-clique Percolation Community (KCPC)). *Given a graph G and an integer k, a k-clique percolation community is the union of a maximal set of k-cliques where any pair of k-cliques in the set is connected.*

Example 1. *Consider the graph G in Figure 1, suppose $k = 4$, there are two KCPCs in G: C_4^1 and C_4^2. The KCPC C_4^1 contains a solitary 4-clique $C_4^1 = \{v_2, v_3, v_7, v_{10}\}$ because there is no other 4-clique sharing three vertices with C_4^1. The KCPC C_4^2 is a set of 4-cliques including $C_4^2 = \{v_2, v_4, v_5, v_7\}$, $C_4^3 = \{v_4, v_7, v_8, v_{11}\}$,*

Table 1: Summary of notations

Notation	Definition
$N_G(u)$	set of neighbors of u in G
$d_G(u)$	degree of u in G
δ	degeneracy of G
C, C_k	$(k\text{-})$clique
$\mathcal{C}$	k-clique percolation community (KCPC), set of cliques
$\mathcal{C}^{Q}$	Quasi-KCPC
MCE	maximal clique enumeration
$T = (ND, \vec{E})$	search tree of MCE
$PF(nd)$	prefix of T, starts at T's root, ends at nd
$nd.PX$	$nd.P \cup nd.X$
nd_{C^i, l_j}	MCE search node at layer l_j of branch C^i
LCC	local connected component
LMC	local maximal clique

$C_4^4 = \{v_7, v_{11}, v_{12}, v_{15}\}$, etc. All 4-cliques in $\mathcal{C}_4^2$ are pairwise connected. For example, the above three cliques can be connected by sequence $C_4^2, \{v_4, v_5, v_7, v_8\}, C_4^3, \{v_7, v_8, v_{11}, v_{15}\}, C_4^4$. The two KCPC*s $\mathcal{C}_4^1$ and $\mathcal{C}_4^2$ share vertices $\{v_2, v_7\}$. Let $k = 3$, $\mathcal{C}_3^1$ is a* KCPC *and contains a solitary 3-clique $C_3 = \{v_{11}, v_{14}, v_{16}\}$ because there is no other 3-clique sharing two vertices with C_3. All 3-cliques in G excluding C_3 form another* KCPC*, and it shares vertex v_{11} with $\mathcal{C}_3^1$.*

Problem statement. Given a graph G and an integer k, the goal of our problem is to identify all KCPCs in G.

From the above definitions, it can be seen that different KCPCs are essentially disjoint sets composed of a set of *connected* k-cliques. Consequently, we are able to use the classic union-find set [8] to maintain the k-clique connectivity of these disjoint sets.

Union-Find Set [8]. The Union-Find Set, also known as the Disjoint-Set data structure, is a highly efficient tool for managing a collection of disjoint sets. Each set is represented as a tree, where nodes correspond to the elements of the set. In this tree structure, each node points to its parent and the root of the tree acts as the representative of the set. This design enables fast **union** and **find** operations, making it particularly well suited for applications such as graph connectivity problems. In the following, we outline the commonly used operations in the Union-Find Set.

Find(i): Gets the representative of the unique set containing i.

Union(i, j): Merges the two sets that contain i and j, respectively, into a single set that is the union of these two sets. Specifically, in the corresponding trees, the parent of the representative of one set is set as the representative of the other set.

Both **union** and **find** operations can be performed in constant amortized time in all practical situations, and the space complexity of the union-find set is $O(n)$ where n is the total number of elements managed by the structure [7, 8].

3 Existing Solutions and Their Defects

Existing solutions on KCPC mining can be categorized into two types: maximal clique enumeration (MCE) based solutions [47, 51, 60] and k-clique listing (KCL) based solutions [5, 33].

3.1 MCE-based Solution

Existing MCE-based solutions rely on the concept of maximal clique adjacency graph. KCPCs can be computed by finding the connected components of maximal clique adjacency graph with the edges whose weights are $\geq k-1$.

Definition 7 (Maximal Clique Adjacency Graph (MCAG)). *Given a graph G, the maximal clique adjacent graph of G, denoted as $G_\phi = (V_\phi, E_\phi, W)$, is a graph with weights. Each of its vertex $u \in V_\phi$ represents a maximal clique C in G and the weight of vertex u is $W(u) = |C|$. For any two vertices $u, v \in V_\phi$, if the two corresponding maximal cliques share $W(u,v)$ vertices with $W(u,v) > 0$, then $(u,v) \in E_\phi$, i.e., the weight of the edge (u,v) is $W(u,v)$.*

Given an MCAG and a path $P_{u_1,u_l} = (u_1, u_2, ..., u_l)$ on it, the weight of P_{u_1,u_l}, denoted by $W(P_{u_1,u_l})$, is the minimum weight of edges in P_{u_1,u_l}, i.e., $W(P_{u_1,u_l}) = \min_{i=1,2,\cdots,l-1} W(u_i, u_{i+1})$. Based on the definition of path weights, we define k-weighted connected component as follows, which is another definition of KCPC.

Definition 8 (k-weighted Connected Component). *Given a graph G, its* MCAG *G_ϕ and an integer k, a k-weighted connected component CC_k of G_ϕ is a maximal vertices set in G_ϕ where $\forall u \in CC_k, W(u) \geq k$ and $\forall u, v \in CC_k, \exists P_{u,v}, W(P_{u,v}) \geq k-1$.*

Theorem 1. *[60] Given a graph G, its* MCAG *G_ϕ and an integer k, a k-weighted connected component in G_ϕ is a* KCPC *in G.*

All existing MCE-based solutions [47, 51, 60] utilize maximal clique enumeration techniques to first construct the Maximal Clique Adjacency Graph (MCAG), and subsequently traverse the MCAG to identify the k-weighted connected components, thereby finding all KCPCs according to Theorem 1. The major drawback of these approaches is that the size of the MCAG can be extremely large. For instance, on a relatively small graph like Duke, which has 9,885 vertices and 506,437 edges, the size of its MCAG can exceed 10^{13} (see Table 2 for details). Consequently, traversing such a massive MCAG to derive all KCPCs is clearly impractical.

3.2 KCL-based Solution

The KCL-based solution was initially proposed in [33]. The basic idea is to list all $(k-1)$-cliques and then connect all k-cliques that share each $(k-1)$-clique. Specifically, the solution in [33] is based on sequentially inserting edges to the network and keeping track of the emerging KCPC. [5] directly lists k-cliques and $(k-1)$-cliques in the complete graph, eliminating the need for dynamic maintenance of KCPC. In the above process, k-cliques can be managed using union-find set. If two k-cliques share a $(k-1)$-clique, then the union-find set connects these two k-cliques into the same KCPC (by merging the KCPC components containing these two k-cliques).

The main limitation of KCL-based solutions is that their time complexity is linearly proportional to the number of k-cliques [5, 33]. This is because these methods require the exhaustive listing of all k-cliques. However, the number of k-cliques grows exponentially as k increases [32, 39, 58], making the task of listing and storing k-cliques computationally infeasible for even very small k (e.g., $k = 7$). Consequently, existing KCL-based solutions become impractical for relatively-large values of k, limiting their applicability in real-world scenarios.

4 A Novel MCE-based Solution

As we discussed in Section 3.1, existing MCE-based methods suffer from the enormous size of MCAG (over 10^7 vertices and 10^{13} edges, see Table 2), resulting in extremely time-consuming traversal of the MCAG. As a result, it can be seen that reducing the size of the MCAG is the key to improving the performance of MCE-based solution.

Key idea. To reduce the size of the MCAG, we leverage the structure of the maximal clique enumeration tree to group maximal cliques during enumeration. This is based on the observation that many branches of maximal cliques in the tree share a prefix of

$k-1$ or more vertices, which corresponds to a $(k-1)$-clique or larger in the graph. These maximal cliques can be pre-determined to belong to the same KCPC and grouped using a union-find structure. After enumeration, each maximal clique belongs to a group, and we construct a group-level MCAG by treating groups as vertices and connecting them based on cross-group adjacency. This grouped MCAG is significantly smaller, as the number of groups is often an order of magnitude less than that of maximal cliques, as shown in our experiments. We call the above groups of maximal cliques as Quasi-KCPCs.

Definition 9 (Quasi-KCPC). *Given a graph G, its maximal clique adjacency graph (MCAG) G_ϕ and an integer k, a Quasi-KCPC is a subset C^{Q} of vertices in G_ϕ where $\forall u \in C^{\mathrm{Q}}, W(u) \geq k$ and $\forall u, v \in C^{\mathrm{Q}}, \exists P_{u,v}, W(P_{u,v}) \geq k-1$.*

Compared to the definition of KCPC (Definition 8), it can be observed that Quasi-KCPC is an incomplete KCPC (a subset of KCPC). Based on Quasi-KCPC, our MCE-based solution can be divided into two stages. In the first stage, we utilize the maximal clique enumeration tree to generate Quasi-KCPCs. In the second stage, we traverse the smaller MCAG based on Quasi-KCPCs to obtain the final KCPCs. Next, we will briefly introduce the maximal clique enumeration tree in Section 4.1, then we introduce the two stages of our MCE-based solution in Section 4.2 and Section 4.3 respectively.

4.1 Maximal Clique Enumeration Tree

The maximal clique enumeration tree is the search tree generated by algorithms for enumerating all maximal cliques. A widely adopted framework is the pivot based recursive approach proposed in [14], which maintains three sets at each search tree node: R, the current clique; P, the set of candidate vertices; and X, the set of excluded vertices to avoid non-maximal results. In each recursive call, a vertex $u \in P$ is selected, and the new sets are defined as $R' = R \cup \{u\}$, $P' = P \cap N_G(u)$, and $X' = X \cap N_G(u)$. All maximal cliques containing R' and vertices in P' but not X' will be derived in this recursive search. After each call, the sets are updated by $P = P \setminus \{u\}$ and $X = X \cup \{u\}$. A search tree node is a leaf if $P = \emptyset$, and R is a maximal clique if $X = \emptyset$ at that leaf. To improve efficiency, a **pivoting strategy** is applied: based on the observation that any maximal clique C within a set S either contains a vertex u or vertices in $S \setminus N(u)$, the algorithm selects a **pivot vertex** $u \in P \cup X$ that has the most neighbors in P, allowing it to skip enumerating most branches. In the root node, P is ordered using degeneracy ordering, yielding $|P| = O(n)$, while in other nodes, $|P| = O(\delta)$, and the height of the enumeration tree is $O(\delta)$, where δ is the degeneracy of the graph.

Example 2. *Figure 2 shows an example MCE tree in [14] on $G = (V, E)$ illustrated in Figure 1. Pivot vertices are enclosed in dashed circles. Some branches are omitted. In the root node, $P = V, X = \emptyset$. There is no need to select a pivot vertex in the root node. Suppose the algorithm are exploring v_2 in the root node (layer 0). $v_3, v_{10}, v_4, v_7, v_5$ are neighbors of v_2, and v_3, v_{10} have being explored, so $X = \{v_3, v_{10}\}, P = \{v_4, v_7, v_5\}, R = \{v_2\}$ in layer 1. v_4 is then selected as the pivot vertex, so that its neighbors, v_7, v_5, will not be explored. Only v_4 will be explored and only one child node is generated in layer 2. $X = \emptyset$ in layer 2 because v_3, v_{10} in layer 1 are not neighbors of v_4. Nodes in the following layers are generated in a similar way.*

Below, we formally define MCE tree.

Definition 10 (MCE Tree). *Given a graph $G = (V, E)$, the MCE tree $T = (ND, \vec{E})$ is the search tree of MCE algorithm on graph G.*

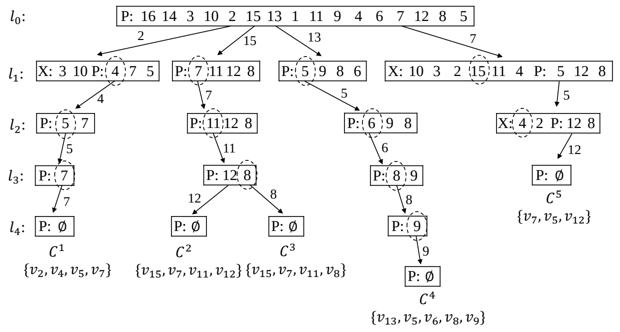


Figure 2: An example maximal clique enumeration tree on graph in Figure 1. Pivot vertices are enclosed in dashed circles. Some branches are omitted. We denote by nd_{C^i,l_j} the MCE search tree node at layer l_j of branch C^i (corresponding to maximal clique C^i)

ND is the set of search tree nodes, which are in form of (P, R, X). For each $nd \in ND$, $nd.P$, $nd.X$ will be updated throughout the algorithm; however, here they remains at its initial state for better understanding. $\vec{E}$ is the set of directed edges in form of $(nd_1, nd_2, u), u \in V$. A directed edge $(nd_1, nd_2, u) \in \vec{E}$ indicates that nd_2 is a child node of nd_1 and the sub-tree rooted at nd_2 is a recursive tree generated based on nd_1 by exploring $u \in nd_1.P$ in the MCE algorithm.

In an MCE tree node nd, the value of $nd.R$ depends on the simple path from the root of the MCE tree to nd, which we define as p-prefix.

Definition 11 (p-prefix). *Given a graph G and the MCE tree $T = (ND, \vec{E})$, a p-prefix $PF(nd_p) = (nd_1, nd_2, ..., nd_p)$ is a path with p nodes in T and nd_1 is the root node of T, $\forall i = 1, 2, ..., p-1, (nd_i, nd_{i+1}, u) \in \vec{E}$.*

Consider an MCE tree $T = (ND, \vec{E})$, we can see that for all $nd \in ND$, $nd.R = \{u | (nd_i, nd_{i+1}, u) \in \vec{E}, nd_i, nd_{i+1} \in PF(nd)\}$, which is the currently identified clique when visiting nd. An MCE tree node nd is a leaf node if $nd.P = \emptyset$, and if a leaf node nd has $nd.X = \emptyset$, then $nd.R$ is a maximal clique. In the following context, $nd.PX$ is short for $nd.P \cup nd.X$.

4.2 Quasi-KCPCs Computation

As mentioned earlier, Quasi-KCPCs are computed based on the structure of the MCE tree. In detail, given an MCE tree $T = (ND, \vec{E})$ and a tree node $nd_p \in ND$, if $PF(nd_p)$ is a p-prefix of T where $p \geq k$, i.e., $|PF(nd_p)| \geq k$, and if the sub-tree rooted at nd_p has branches representing maximal cliques (these branches end at leaf nodes nd where $nd.PX = \emptyset$, and $nd.R$ are those maximal cliques), then these maximal cliques must belong to the same KCPC, as they must share a $(k-1)$-clique C_{k-1} that $C_{k-1} \subseteq nd_p.R$.

Example 3. *Consider the MCE tree in Figure 2, let $k = 3$, C^2 and C^3 are in the same KCPC because these two branches share p-prefix $PF(nd_{C^2,l_2})$ and $|PF(nd_{C^2,l_2})| \geq k$ (C^2, C^3 share $nd_{C^2,l_2}.R = \{v_7, v_{15}\}$).*

As a result, making use of p-prefix $PF(nd_p)$ ($p \geq k$), we can connect maximal cliques sharing $nd_p.R$ into one same Quasi-KCPC. The above p-prefixes can be divided into two types: $nd_p.X = \emptyset$ and $nd_p.X \neq \emptyset$.

Type I: $nd_p.X = \emptyset$. This case is relatively simple and corresponds to the situation described in Example 3. Specifically, $nd_p.X = \emptyset$ indicates that for all maximal cliques sharing $nd_p.R$, their corresponding branches are located within the sub-tree rooted at nd_p, or share prefix $PF(nd_p)$. Quasi-KCPC can be obtained by connecting all the maximal clique branches in the sub-tree using union-find set.

Type II: $nd_p.X \neq \emptyset$. If $nd_p.X \neq \emptyset$, there are branches of maximal cliques sharing $nd_p.R$ that do not reside within the sub-tree rooted at nd_p, because these maximal cliques contain vertex $u \in nd_p.X$, and have been derived in branches generated in exploring u. For example, let $nd_p = nd_{C^5,l_2}$ in Figure 2, the branch of maximal clique C^1 does not reside within the sub-tree rooted at nd_{C^5,l_2}, while C^1 shares $nd_{C^5,l_2}.R = \{v_5, v_7\}$.

We need to take into account nd_p of both Type I and Type II, because our goal is to ensure that the average size of final Quasi-KCPCs is as large as possible, so that the number of Quasi-KCPCs needed to cover all maximal cliques is minimized, thus reducing the size of MCAG.

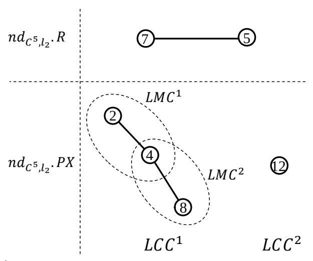
Challenge to handle p-prefixes of Type II. For $PF(nd_p)$ of Type II, to find the maximal clique branches that share $nd_p.R$ but are outside the sub-tree rooted at nd_p, a straightforward approach is to traverse the MCE tree in order to identify those maximal clique branches that share $nd_p.R$. However, the traversal space can be quite large, since the size of the MCE tree and the number of maximal clique branches can be very large ($O(\delta n 3^{\frac{\delta}{3}})$ and $O((n-\delta)3^{\frac{\delta}{3}})$ respectively [13], n is the number of graph vertices, δ is the degeneracy of the graph). Performing the above operations for every $PF(nd_p)$ is impractical.

To address the above problem, we treat the MCE tree as an index and use attributes of each tree node nd_p as query parameters, transforming the problem into a series of maximal clique branch queries. Note that we do not store the entire MCE tree—this is further discussed in Section 4.3 when analyzing the overall complexity of Algorithm 3. Specifically, we use the vertex sets $nd_p.R$ and $nd_p.PX$ as query parameters. Since $nd_p.PX$ contains all common neighbors of $nd_p.R$, any maximal clique containing $nd_p.R$ must be formed by combining $nd_p.R$ with a local maximal clique in the subgraph induced by $nd_p.PX$. To avoid enumerating all local maximal cliques in the induced subgraph, we partition it into connected components. Each component, together with $nd_p.R$, is used to identify a maximal clique branch, and all maximal cliques identified in this way are connected to the same Quasi-KCPC. Moreover, when processing another node nd with $nd_p.R \subset nd.R$, the vertices in $nd.R \setminus nd_p.R$ may further divide these components, connecting previously unconnected maximal cliques into the same Quasi-KCPC.

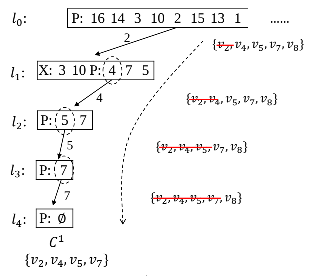
Next, we present the detailed definitions and algorithms.

Definition 12 (Local Connected Component (LCC) and Local Maximal Clique (LMC)). *Given a graph $G = (V, E)$, $V' \subseteq V$ and $G_{V'}$ is the subgraph of G induced by V', a local connected component $LCC \subseteq V'$ is a maximal vertices set that each pair of vertices in LCC are connected in $G_{V'}$. A local maximal clique $LMC \subseteq V'$ is a maximal clique in $G_{V'}$.*

Example 4. *In Figure 3 (a), LCC^1, LCC^2 are two local connected components in $nd_{C^5,l_2}.PX$ (nd_{C^5,l_2} is the tree node at layer l_2 of branch C^5 in Figure 2). There are two local maximal cliques LMC^1, LMC^2 in LCC^1, while there is only a single vertex in LCC^2.*



(a) Local connected component (LCC) and local maximal clique (LMC) in $nd_{C^5,l_2}.PX$



l_0:
P: 16 14 3 10 2 15 13 1
......
2
$\{\sout{v_2,} v_4, v_5, v_7, v_8\}$
l_1:
X: 3 10 P: 4 7 5
4
$\{\sout{v_2, v_4}, v_5, v_7, v_8\}$
l_2:
P: 5 7
5
$\{\sout{v_2, v_4, v_5}, v_7, v_8\}$
l_3:
P: 7
7
$\{\sout{v_2, v_4, v_5, v_7}, v_8\}$
l_4:
P: ∅
C^1
$\{v_2, v_4, v_5, v_7\}$

(b) Identify branch C^1 of MCE tree T with $Y = nd_{C^5,l_2}.R \cup LCC^1$ ($\{v_2, v_4, v_5, v_7, v_8\}$)

Figure 3: Illustration of LCC, LMC and process in Algorithm 1. T is the MCE tree in Figure 2, nd_{C^5,l_2} is the tree node at layer l_2 of branch C^5 in T

Algorithm 1: IdentifyBranch (T, Y)

```
Input: An MCE tree $T = (ND, \vec{E})$; A vertex set $Y$
Output: Branch of a maximal clique in $Y$
1 Let $nd$ be the root of $T$;
2 while $true$ do
3     if $nd.X \cup nd.P = \emptyset$ then
4         return BranchId ($nd$); /* It can be the subscript of $nd.R$ in maximal clique
              array where it is stored */
5     Let $childs \leftarrow \{nd_c | (nd, nd_c, *) \in \vec{E}\}$ be the list of child nodes of $nd$ and
          ordered by their creation time;
6     for $nd_c \in childs$ do
7         if $\nexists nd_l$ in the sub-tree rooted at $nd_c$ that $nd_l.X \cup nd_l.P = \emptyset$ then continue;
8         if $(nd_c.R - nd.R) \subseteq Y$ then
9             $nd \leftarrow nd_c$; break; /* $nd_c.R$ has only one more item than $nd.R$ */
```

As introduced earlier, we use a local connected component $LCC \subseteq nd_p.PX$ and $nd_p.R$ to form the query set $Y = nd_p.R \cup LCC$. Algorithm 1 then returns the branch in the MCE tree T corresponding to a maximal clique containing $nd_p.R$. The search starts from the root (Line 1). If a node nd is a leaf with $nd.P \cup nd.X = \emptyset$, the branch is returned (Line 3-4). Otherwise, child nodes are traversed in creation order (Lines 5–6). If a subtree rooted at nd_c contains no maximal clique, it is skipped (Line 7). For each valid nd_c, the difference $nd_c.R - nd.R$ contains a single vertex u, indicating that nd_c was created by exploring $u \in nd.P$. If $u \in Y$, the search descends into nd_c (Line 9), continuing until the branch is found.

Theorem 2. *Given a p-prefix $PF(nd_p)$ in MCE tree $T = (ND, \vec{E})$ and a local connected component $LCC \subseteq nd_p.PX$, Algorithm 1 is capable of returning the branch of a maximal clique formed by $nd_p.R$ and a local maximal clique $LMC \subseteq LCC$.*

Proof. Let $Y = nd_p.R \cup LCC$, we first prove that Algorithm 1 returns the branch of a maximal clique formed by vertices in Y. If the condition in Line 3 is satisfied, then the algorithm finds a maximal clique in Y since each vertex in $nd.R$ is verified to be in the set Y in Line 8. If the condition in Line 3 is not satisfied, then the algorithm traverses all child nodes of nd and there must be a child node nd_c that satisfies the condition in Line 8. It can be proven by contradiction. Suppose the current visited prefix in T is $PF(nd_{p'})$ and there is no child node of $nd_{p'}$ that satisfies the condition in Line 8. It is clear that $nd_{p'}.R \subset Y$ is not a maximal clique. For any maximal clique $C \subseteq Y$ and $nd_{p'}.R \subset C$, suppose $LV = C - nd_{p'}.R$ and according to the MCE algorithm, $LV \subseteq nd_{p'}.PX$. Since there is no child node of $nd_{p'}$ that satisfies the condition in Line 8, there must be

Algorithm 2: Quasi-KCPC (G, T, k)

```
   Input: Graph $G = (V, E)$; The MCE tree $T = (ND, \vec{E})$; An integer $k$
   Output: The union-find set $UF$ that records all the Quasi-KCPCs
1  $UF \leftarrow$ an initially empty union-find set;
2  for $p$-prefix $PF(nd_p)$ in $T$ and $p \geq k$ do
3      $PX \leftarrow nd_p.P \cup nd_p.X$;
4      Let $LCC^1, LCC^2, ..., LCC^l$ be the local connected components in $PX$;
5      if $l = 1$ then continue;
6      $ids \leftarrow \emptyset$;
7      for $i \leftarrow 1$ to $l$ do
8          $Y \leftarrow nd_p.R \cup LCC^i$;
9          $cid \leftarrow$ IdentifyBranch $(T, Y)$; $ids \leftarrow ids \cup \{cid\}$;
10     Let $cid_0 \in ids$;
11     for $cid \in ids$ do $UF.union(cid, cid_0)$ ;
12 return $UF$;
```

a set of vertices $LX \subseteq LV$ that $LX \subseteq nd_{p'}.X$. According to the MCE algorithm, there must be some nodes nd_i in $PF(nd_{p'})$ that $LX \subseteq nd_i.P$ (e.g., $LX \subseteq nd_1.P$ where nd_1 is the root). Assume that nd_i is the last node that $LX \subseteq nd_i.P$, i.e., $\exists u \in LX, u \in nd_{i+1}.X$. In this case, Algorithm 1 must have skipped all $(nd_i, *, u) \in \vec{E}, u \in LX$ so that u will be in $nd_{i+1}.X$, which contradicts the actual order where the algorithm visits the child nodes of nd_i. As a result, Algorithm 1 can find a maximal clique in Y, and a maximal clique in Y must be formed by $nd_p.R$ and a local maximal clique in LCC, since $Y = nd_p.R \cup LCC$, $nd_p.R$ is already a clique and vertices in LCC are shared neighbors of vertices in $nd_p.R$. □

Example 5. *Figure 3 (b) illustrates the process of identifying branch C^1 with $Y = nd_{C^5,l_2}.R \cup LCC^1$. Based on Figure 3 (a), $Y = \{v_2, v_4, v_5, v_7, v_8\}$. For the root node nd_{C_1,l_0} of T, nd_{C_1,l_1} is the first child node that satisfies $(nd_{C_1,l_1}.R - nd_{C_1,l_0}.R) \subseteq Y$, so nd in Line 9 of Algorithm 1 is set to nd_{C_1,l_1} and the loop in Line 2 continues. Subsequently, the entire branch C^1 will be identified, and $C^1 = nd_{C^5,l_2}.R \cup LMC^1$.*

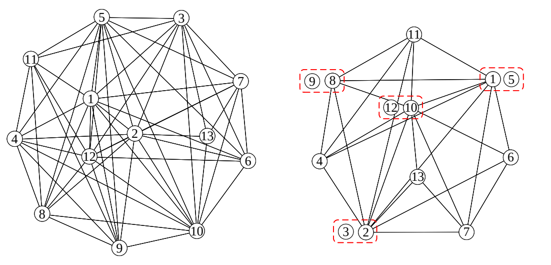
Theorem 3. *The time complexity of Algorithm 1 is $O(n + \delta^2)$, where n, δ are the number of vertices and the degeneracy of the original graph respectively.*

Proof. For the root node and the other nodes of T, there are at most $O(n)$ child nodes and $O(\delta)$ child nodes respectively [14]. In the worst case, Algorithm 1 will traverse to the last nd_c in Line 6 for each nd. Since the depth of T is bounded by $O(\delta)$, the time complexity of Algorithm 1 is $O(n + \delta^2)$. □

The time complexity of Algorithm 1 is derived under the worst-case assumption. We assume that the root node of the MCE tree has n child nodes, each other node has δ child nodes, and Algorithm 1 will traverse to the last nd_c in Line 6 for each nd. However, such assumptions are rarely valid in practical scenarios.

Combining Algorithm 1, we present Algorithm 2 to compute Quasi-KCPC in a graph. The algorithm traverses all p-prefixes of T where $p \geq k$ (Line 2) and collects local connected components in $nd_p.P \cup nd_p.X$ (Line 4). If there is only one local connected component, then the current $PF(nd_p)$ is skipped (Line 5). In Lines 7-9, for each LCC^i, the algorithm utilizes Algorithm 1 to obtain the branch of a maximal clique in $nd_p.R \cup LCC^i$. In Lines 10-11, the algorithm connects all maximal cliques in ids into the same Quasi-KCPC using a union-find set UF.

Theorem 4. *Algorithm 2 correctly computes the* Quasi-KCPC*s.*



(a) The MCAG, with 13 vertices and 57 edges (b) The MCAG based on Quasi-KCPC, with 9 Quasi-KCPCs and 25 edges

Figure 4: The comparison between the original MCAG **and** MCAG **based on** Quasi-KCPC, $k = 3$. **Weights on vertices and edges are omitted.** Quasi-KCPC**s with two or more vertices in (b) are enclosed in dashed rectangles**

Proof. In Line 9, by Theorem 2, IdentifyBranch is guaranteed to find a maximal clique containing $nd_p.R$. Moreover, in Line 2, $PF(nd_p)$ is a p-prefix with $p \geq k$, ensuring that the maximal cliques connected in Line 11 belong to the same KCPC, thus satisfying the definition of Quasi-KCPC. □

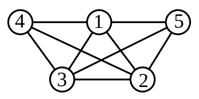
Theorem 5. *The time complexity of Algorithm 2 is $O((n + \delta^2)\delta n^2 3^{\frac{\delta}{3}})$.*

Proof. Based on [13, 14], the time complexity of MCE algorithm in [14] without reporting maximal cliques is $O(\delta n 3^{\frac{\delta}{3}})$, thus, the number of tree nodes in T is bounded by $O(\delta n 3^{\frac{\delta}{3}})$. For each $PF(nd_p)$, $|PX|$ in Line 3 is bounded by $O(n)$, and the time complexity of computing the local connected components in PX is $O(n\delta)$ since the number of edges in the subgraph induced by PX is bounded by $n\delta$ [14]. The number of local connected components in PX is also bounded by $O(n)$, thus, the time complexity of Lines 7-9 is $O(n(n + \delta^2))$. Overall, the time complexity of Algorithm 2 is $O((n\delta + n(n + \delta^2))\delta n 3^{\frac{\delta}{3}}) = O((n + \delta^2)\delta n^2 3^{\frac{\delta}{3}})$. □

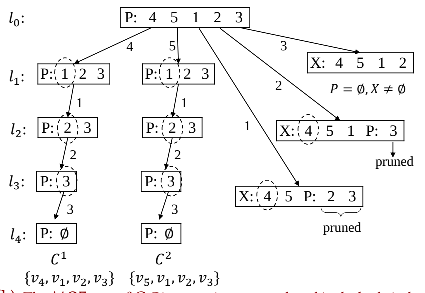
The time complexity of Algorithm 2 comprises a polynomial part and an exponential part. The exponential complexity arises from the inherent complexity of maximal clique enumeration [14], where δ is usually small in real-world graphs. For the polynomial part, $(n+\delta^2)$ comes from Algorithm 1, while δn^2 includes one n from maximal clique enumeration [14] and another n to bound $|PX|$. Consider nd_p in Line 3, it is highly unlikely for PX to have all n vertices in real-world graphs to be common neighbors of $nd_p.R$. Therefore, the stated time complexity is rarely reached, and Algorithm 2 is efficient in practical scenarios.

Example 6. *Let $k = 3$, Figure 4 (a) shows the* MCAG *of graph G in Figure 1. Recall the* MCE *tree in Figure 2 and based on Algorithm 2, C^2, C^3 and C^1, C^5 can be connected into one* Quasi-KCPC *respectively. Considering the maximal cliques connected through p-prefixes within the hidden branches in Figure 2, we can obtain the* MCAG *based on* Quasi-KCPC*s in Figure 4 (b). In Figure 4 (b),* Quasi-KCPC*s are considered as "super vertices", so that a smaller-scale* MCAG *can be constructed.*

Quantifying Quasi-KCPC **as an approximation of** KCPC. We can see that the Quasi-KCPCs obtained are not the final KCPCs. The underlying cause of this phenomenon is the pivot selection in the MCE algorithm. Specifically, the pivot selection technique



(a) A simple graph G



(b) The MCE tree of G. Pivot vertices are enclosed in dashed circles

Figure 5: Case study of Theorem 6. $k = 4$

ensures that certain vertices in P set of each search tree node are not enumerated, thereby preventing the formation of sufficient p-prefix, and causing some maximal cliques to remain unconnected.

Given a simple graph G and an integer k, suppose n is the number of vertices in G, n_{clique} is the number of maximal cliques in G with size at least k, δ is the degeneracy of G. Let $\mathcal{Q}$ be the number of Quasi-KCPCs and $\mathcal{K}$ be the number of KCPCs, we present three theorems that characterize how pivot selection affects the approximation between Quasi-KCPC and KCPC, which is measured by $\frac{\mathcal{Q}}{\mathcal{K}}$.

Theorem 6. *If pivot selection is applied to all nodes in the* MCE *search tree, then* $\frac{\mathcal{Q}}{\mathcal{K}} \leq n_{clique} \leq (n - \delta)3^{\delta/3}$*, and there exists a case where* $\frac{\mathcal{Q}}{\mathcal{K}} = n_{clique}$.

Proof. $n_{clique} \leq (n - \delta)3^{\delta/3}$ is given by [13]. $\frac{\mathcal{Q}}{\mathcal{K}} \leq n_{clique}$ can be rewritten as $\mathcal{Q} \leq \mathcal{K} n_{clique}$. It is straightforward to see that $\mathcal{Q} \leq n_{clique}$, since Quasi-KCPCs are sets of maximal cliques, and Quasi-KCPCs are maintained by the Union-Find Set, ensuring that Quasi-KCPCs do not overlap. Therefore, $\mathcal{Q} \leq n_{clique}$. Additionally, since the number of KCPCs is greater than 0, thus $\mathcal{Q} \leq \mathcal{K} n_{clique}$. □

Case study of Theorem 6. Consider the graph G in Figure 5 (a) with $k = 4$, which contains two maximal cliques, $\{v_4, v_1, v_2, v_3\}$ and $\{v_5, v_1, v_2, v_3\}$, forming a single KCPC as they share $k - 1 = 3$ vertices. Figure 5 (b) shows the complete MCE tree. The root node uses degeneracy ordering $P = \{v_4, v_5, v_1, v_2, v_3\}$, and the tree enumerates both maximal cliques through branches C_1 and C_2. When branching from v_1, the pivot is v_4 or v_5, each having two neighbors in $P = \{v_2, v_3\}$, while v_2 and v_3 have only one neighbor in P. Thus, branches from v_2 and v_3 are pruned. As a result, the shared vertex set $\{v_1, v_2, v_3\}$ does not appear as a prefix in the tree, and the two maximal cliques form separate Quasi-KCPCs. Hence, $\mathcal{Q} = 2$, $\mathcal{K} = 1$, and $n_{\text{clique}} = 2$.

Theorem 7. *If pivot selection is not applied to any nodes in the* MCE *search tree, then* $\frac{\mathcal{Q}}{\mathcal{K}} = 1$.

Proof. Let $T = (ND, \vec{E})$ be the MCE tree. Without pivot selection, the enumeration explores the power set of vertices in G, and for any two maximal cliques C^1, C^2, there exists a p-prefix $PF(nd)$ such that $nd.R = C^1 \cap C^2$. The proof proceeds in two steps. First, we show that for each prefix PF_{nd_p} with $p \geq k$, Algorithm 2 connects all maximal cliques sharing $nd_p.R$ into one Quasi-KCPC. Let ND_k be the set of nodes $nd_p \in ND$ with $p \geq k$ and multiple local connected components (two or more) in $nd_p.PX$. Let $\{r_1, r_2, \ldots, r_m\}$ be the sizes of $nd.R$ in ND_k, ordered by $r_i < r_{i+1}$. For the largest r_m, each local connected component in $nd_{r_m}.PX$ corresponds to a local maximal clique; otherwise, a larger r_{m+1} would exist. Thus, Algorithm 2 connects all maximal cliques sharing $nd_{r_m}.R$. By induction, assume the result holds for $r_j, r_{j+1}, \ldots, r_{m-1}$, and consider $nd_{r_{j-1}}$. For any two adjacent local maximal cliques $LMC^1, LMC^2 \subseteq LCC \subseteq nd_{r_{j-1}}.PX$, there exists nd_{r_i} with $i > j - 1$ and $nd_{r_i}.R = nd_{r_{j-1}}.R \cup (LMC^1 \cap LMC^2)$. Thus, maximal cliques within $nd_{r_{j-1}}.R \cup LCC$ are connected. By Algorithm 2, each LCC produces a maximal clique identified in Line 9, and these cliques are connected in Line 11. Thus, the induction completes. Second, we prove that the Quasi-KCPCs obtained by Algorithm 2 are exactly KCPCs in G. Suppose by contradiction that a Quasi-KCPC $C^{\mathcal{Q}}$ obtained by Algorithm 2 is strictly contained in a KCPC CC_k. Since maximal cliques in CC_k are connected, there must exist $C^1 \in CC_k \setminus C^{\mathcal{Q}}$ and $C^2 \in C^{\mathcal{Q}}$ such that $|C^1 \cap C^2| \geq k - 1$. Without pivoting, there must be a p-prefix PF_{nd_p} with $nd_p.R = C^1 \cap C^2$, connecting C^1 and C^2 in Algorithm 2. Hence, $C^{\mathcal{Q}} = CC_k$, and $\frac{\mathcal{Q}}{\mathcal{K}} = 1$. □

In the algorithm implementation, we can use a parameter l to control the number of search tree nodes where pivot selection is applied, i.e., for a node nd, if $|nd.P| \leq l$, then the pivot technique is not applied. Theorem 6 corresponds to cases where $l = 0, 1$, while Theorem 7 corresponds to cases where $l \geq \delta$ (for every search node in the MCE tree, the size of the candidate set $|P| \leq \delta$ [13]). For cases where l lies between 1 and δ, larger l results in more MCE tree branches and increase the chance for more maximal cliques to find their p-prefix ($p \geq k$), making the Quasi-KCPCs computed in Algorithm 2 more closer to KCPCs, which is reflected in the following theorem.

Theorem 8. *For* l_1 *and* l_2*, where* $0 \leq l_1 < \delta$ *and* $l_2 = l_1 + 1$*, let* $\frac{\mathcal{Q}_1}{\mathcal{K}}, \frac{\mathcal{Q}_2}{\mathcal{K}}$ *be the ratios of the number of* Quasi-KCPC*s to the number of* KCPC*s for* l_1 *and* l_2*, respectively. Then,* $\frac{\mathcal{Q}_1}{\mathcal{K}} \geq \frac{\mathcal{Q}_2}{\mathcal{K}}$.

Proof. We only need to prove that $\mathcal{Q}_1 \geq \mathcal{Q}_2$. When $l = l_1$, recall that in the MCE tree, all maximal cliques have already been enumerated. When $l = l_2$, the MCE tree will generate additional branches, but these branches do not contain any new maximal cliques. In the process of computing Quasi-KCPCs in Algorithm 2, only branches containing maximal cliques are identified by Algorithm 1. Therefore, if the new branches introduced by increasing l are disregarded, the Quasi-KCPC remains unchanged. However, the newly generated branches may also produce new p-prefixes where $p \geq k$, causing more maximal cliques to be merged into the same Quasi-KCPC. This makes the Quasi-KCPCs undergo further merging on the original basis, bringing them closer to the KCPCs, and thus the number of Quasi-KCPCs may either decrease or remain the same ($\mathcal{Q}_1 \geq \mathcal{Q}_2$). □

The analysis above demonstrates that increasing l improves the approximation of Quasi-KCPC to KCPC. However, larger l also expands the search space, leading to low efficiency. Detailed experimental results will be presented in Section 7.

4.3 From Quasi-KCPC to KCPC

Since the maximal cliques in a Quasi-KCPC must belong to the same KCPC, the adjacency relationships between maximal cliques in the same Quasi-KCPC do not need to be traversed. Instead, only the adjacency relationships between maximal cliques from different Quasi-KCPCs need to be explored, and we do not need to actually store a MCAG based on Quasi-KCPC. For two Quasi-KCPCs

Algorithm 3: Maximal clique based solution (QKC)

```
Input: A graph $G$; An integer $k$
Output: All KCPCs
1 $C \leftarrow$ MaximalCliqueEnumeration $(G, k)$; /* $C$ is an array of maximal cliques in $G$ larger than $k-1$ */
2 Let $T$ be the MCE tree;
3 $UF \leftarrow$ Quasi-KCPC $(G, T, k)$; /* Obtain Quasi-KCPC using Algorithm 2 */
4 for $cid \leftarrow 0$ to $C.size() - 1$ do
5     for $cid'$ that $cid', cid$ are in different Quasi-KCPC do
6         if $|C[cid] \cap C[cid']| \geq k-1$ then $UF.union(cid', cid)$ ;
7 return $UF$;
```

$C^{Q,1}, C^{Q,2}$, if $\exists C^1 \in C^{Q,1}, C^2 \in C^{Q,2}$ that $|C^1 \cap C^2| \geq k-1$, then $C^{Q,1}, C^{Q,2}$ can be merged since they belong to the same KCPC.

Algorithm 3 shows the complete process of mining KCPC. It first enumerates all maximal cliques larger than $k-1$ in Line 1, and extract the MCE tree in Line 2. Then Algorithm 2 computes Quasi-KCPC in Line 3. In Lines 4-6, for each maximal clique represented by cid, the algorithm only traverses maximal cliques in different Quasi-KCPCs from where cid is. After that, all KCPCs are recorded in UF (Line 7).

Theorem 9. *Algorithm 3 correctly computes all* KCPC*s.*

Proof. It can be proven by contradiction. Suppose one of the results of Algorithm 3 is a Quasi-KCPC C^Q but not KCPC and C^Q is contained in a KCPC C, there must exist two maximal cliques C^1, C^2 that $|C^1 \cap C^2| \geq k-1, C^1 \in C^Q, C^2 \in C - C^Q$. However, C^1, C^2 must have been traversed in Line 3 of Algorithm 3 and been connected into one same KCPC, thus $C^2 \in C^Q$, which contradicts the precondition. □

Note that in Lines 1-3, MCE and Quasi-KCPC computation can be conducted simultaneously, because the process of enumerating maximal cliques essentially traverses the MCE tree. The MCE tree does not need to be fully stored, because branches in the MCE tree that do not lead to maximal cliques are not needed in maximal clique branch identification in Algorithm 1. Based on this, we present the complexity of Algorithm 3.

Theorem 10. *The time and space complexity of Algorithm 3 is $O(T_{clique} + T_{qkc} + (1 - \frac{1}{n_{qkc}})n_{clique}^2\delta)$ and $O(n_{clique}\delta)$ respectively, where T_{clique}, T_{qkc} are time complexity of maximal clique enumeration and Algorithm 2. n_{qkc} is the number of* Quasi-KCPC*s, n_{clique} is the number of maximal cliques and δ is the degeneracy of the graph.*

Proof. In the worst case, Algorithm 3 only skips the inner mutual visits of maximal cliques in each initial Quasi-KCPCs in Lines 4-6. Let $x = n_{clique}, y = n_{qkc}$ and $n_1, n_2, ..., n_y$ be the number of maximal cliques in each Quasi-KCPC, the time complexity of Lines 4-6 is $O(x^2\delta - (n_1^2 + n_2^2 + ... + n_y^2)\delta) = O(x^2\delta - (\frac{x}{y})^2 y\delta)$ (based on Cauchy–Schwarz inequality), which can be written as $O((1 - \frac{1}{n_{qkc}})n_{clique}^2\delta)$. Since Algorithm 3 includes maximal clique enumeration and Quasi-KCPC computation, the time complexity of Algorithm 3 is $O(T_{clique} + T_{qkc} + (1 - \frac{1}{n_{qkc}})n_{clique}^2\delta)$.

It is clear that the space complexity is bounded by the number of maximal cliques and MCE tree. Due to the fact that Quasi-KCPC computation can be performed simultaneously with the MCE process, the P, X sets of MCE tree nodes do not need to be persistently stored in memory. Additionally, only the branches of the MCE tree that lead to maximal cliques are retained. As a result, the space complexity of Algorithm 3 is $O(n_{clique}\delta)$. □

Algorithm 4: KCL-based solution (KCL-MCL)

```
Input: A graph $G = (V, E)$; An integer $k$
Output: All KCPCs
1 $C \leftarrow$ MaximalCliqueEnumeration $(G, k)$; /* $C$ is an array of maximal cliques in $G$ larger than $k-1$ */
2 Let $UF$ be a union-find set in initial state of size $|C|$;
3 Let $\vec{G} = (V, \vec{E})$ be a DAG generated by any vertex ordering on $G$;
4 ListConnectMClique $(\vec{G}, k, \emptyset)$;
5 return $UF$;

6 Procedure ListConnectMClique $(\vec{G} = (V, \vec{E}), k, R)$
7 $RC \leftarrow \{C | R \subseteq C, C \in C\}$; /* maximal cliques sharing $R$ */
8 if $|\{UF.find(C) | C \in RC\}| \leq 1$ then return; /* there is only 1 or 0 $UF$ disjoint set in $RC$ */
9 if $|R| = k-1$ then
10     Connect all maximal cliques in $RC$ using $UF$;
11 else
12     for $u \in \vec{G}$ do
13         Let $\vec{G}_u$ be the subgraph of $\vec{G}$ induced by all out-going neighbors of $u$;
14         ListConnectMClique $(\vec{G}_u, k, R \cup \{u\})$;
```

In practical scenarios, smaller Quasi-KCPCs can merge into larger Quasi-KCPCs in Algorithm 3, allowing an increasing number of inner mutual visits in Quasi-KCPC to be skipped. As a result, the actual complexity of Algorithm 3 is often significantly lower than the theoretical upper bound.

5 Novel KCL-based Solutions

As we discussed in Section 3.2, traditional KCL-based methods can not handle the exponential growth in the number of k-cliques and the corresponding search space as k increases. Therefore, it is necessary to prune the k-clique listing process to accomplish the KCPC mining task. The first step is to change the connection target of $(k-1)$-clique. Instead of connecting k-cliques sharing $(k-1)$-clique in traditional KCL-based methods, we connect maximal cliques sharing $(k-1)$-clique. Similarly, we still use union-find set to connect maximal cliques that share the same $(k-1)$-clique. Next, we provide the theoretical basis for the pruning operation.

Theorem 11. *Given a graph G, an integer k and the set of all maximal cliques in G, C. Let $k_1 < k_2 \leq k-1$, for a k_1-clique C_{k_1}, if maximal cliques sharing C_{k_1} (i.e., maximal cliques in $\{C | C_{k_1} \subset C, C \in C\}$) are already in the same* KCPC *component, then for any k_2-clique C_{k_2} ($C_{k_1} \subset C_{k_2}$), all maximal cliques sharing C_{k_2} ($\{C | C_{k_2} \subset C, C \in C\}$, if they exist) must also be in the same* KCPC *component.*

Proof. Let $C_1 = \{C | C_{k_1} \subset C, C \in C\}, C_2 = \{C | C_{k_2} \subset C, C \in C\}$, it is clear that $C_2 \subseteq C_1$, because $C_{k_1} \subset C_{k_2}$. Then, if maximal cliques in C_1 are in the same KCPC component, maximal cliques in C_2 must also be in the same KCPC component. □

Based on Theorem 11, our methods enumerate all maximal cliques in the first step. Then, during the process of listing $(k-1)$-cliques, if a k_1-clique C_{k_1} ($k_1 < k-1$) is found such that all maximal cliques sharing C_{k_1} are already in the same KCPC, then all subsequent enumeration branches can be pruned, since all $(k-1)$-cliques containing C_{k_1} can not connect two or more KCPC components

Algorithm 4 implements our KCL-based method that prunes the search space of k-clique listing with original maximal cliques. For k-clique listing, Algorithm 4 adopts the ordering based k-clique listing algorithms [9, 39] as its framework. Algorithm 4 first enumerates all maximal cliques larger than $k-1$ in Line 1. In Line 2-3, it initializes the union-find set UF and the ordered graph $\vec{G}$. Procedure ListConnectMClique lists $(k-1)$-cliques and connect maximal

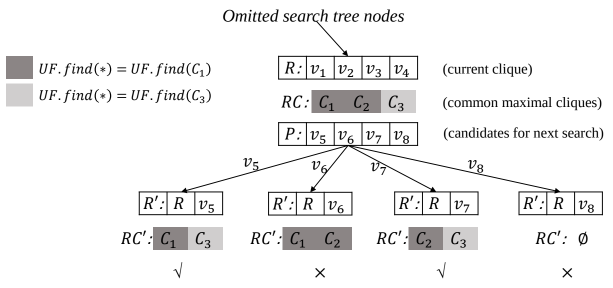


Figure 6: Illustration of k-clique listing with our pruning strategy. Different colors in RC, RC' represent disjoint sets in union-find (UF) set structure. Child nodes with only one color or no color in RC' will be pruned

cliques sharing the same $(k-1)$-clique. It computes all maximal cliques sharing R in Line 7, which is the current listed clique. If there is only one or no UF disjoint set in RC (Line 8), the current branch can be pruned. Otherwise, in Line 9, if R is already a $(k-1)$-clique, then all maximal cliques in RC will be connected together by UF (Line 10). In Lines 12-14, the algorithm continues to expand the current clique R from $\vec{G}$, thereby enumerating larger cliques.

Example 7. *Figure 6 illustrates the pruning strategy in the k-clique listing process using a parent node and four child nodes. In the parent node, R is the current clique, and RC is the set of common maximal cliques of vertices in R, with different colors indicating their union-find sets. The candidate set P includes vertices for expansion, each corresponding to a child node.*

In each child, $R' = R \cup \{v\}, v \in P$, and RC' is the updated set of common maximal cliques. If RC' spans multiple union-find sets (e.g., child after exploring v_5), the search proceeds. If all cliques in RC' belong to one set or fewer (e.g., child after exploring v_6, v_8), the branch is pruned.

Theorem 12. *Algorithm 4 correctly finds all* KCPC*s in G.*

Theorem 12 can be proven using methods similar to proof of Theorem 9.

Theorem 13. *Let V_o be the set of vertices shared by any two maximal cliques, the time and space complexity of Algorithm 4 is $O(T_{clique} + m_o n_{clique}(\frac{\delta_o}{2})^{k-3})$ and $O(n_{clique}\delta)$ respectively, where T_{clique} is the time used in* MCE *algorithm, m_o, δ_o are number of edges and degeneracy of subgraph induced by V_o.*

Proof. Based on [39], the number of k-clique enumeration tree nodes are bounded by $O(m(\frac{\delta}{2})^{k-2})$. For any vertex u in $V - V_o$ that is traversed in Line 12, the condition in Line 8 of the subsequent recursive call is always satisfied. As a result, only vertices in V_o and subgraph induced by V_o should be concerned. In Lines 7-8, $|RC|$ is bounded by $O(n_{clique})$, so that the time complexity of ListConnectMClique is $O(m_o n_{clique}(\frac{\delta_o}{2})^{k-3})$, and the time complexity of Algorithm 4 is $O(T_{clique} + m_o n_{clique}(\frac{\delta_o}{2})^{k-3})$.

Since Algorithm 4 does not need to store all $(k-1)$-cliques and the memory is mainly consumed by maximal cliques, so the space complexity of Algorithm 4 is $O(n_{clique}\delta)$. □

In practical scenarios, RC in Line 7 is unlikely to include all maximal cliques, and the $(k-1)$-clique search tree will be significantly pruned. As a result, Algorithm 4 is hard to reach its upper bound on time complexity.

Algorithm 5: KCL-based solution (KCL-QKC)

```
Input: A graph $G = (V, E)$; An integer $k$
Output: All KCPCs
1 $C \leftarrow$ MaximalCliqueEnumeration $(G, k)$; /* $C$ is an array of maximal cliques in $G$
    larger than $k-1$ */
2 Let $UF$ be a union-find set in initial state of size $|C|$;
3 Let $\vec{G} = (V, \vec{E})$ be a DAG generated by any vertex ordering on $G$;
4 Let $T$ be the MCE tree; /* the difference between Algorithm 5 and Algorithm 4 */
5 Quasi-KCPC $(G, T, k, UF)$; /* the difference between Algorithm 5 and
    Algorithm 4 */
6 ListConnectMClique $(\vec{G}, k, \emptyset)$; /* procedure in Algorithm 4 */
7 return $UF$;
```

Pruning with Quasi-KCPC**.** As introduced in Section 4, Quasi-KCPC represents a group of maximal cliques that are already determined to belong to the same KCPC. Therefore, leveraging Quasi-KCPC can further enhance the pruning effect by enabling earlier pruning operations than original maximal cliques. Based on this, we can directly develop Algorithm 5.

The only difference of Algorithm 5 compared with Algorithm 4 is that Algorithm 5 uses Quasi-KCPCs to further prune the search tree of k-clique listing. In Lines 4-5, Algorithm 5 extracts the MCE tree T and computes Quasi-KCPCs. In Line 6, the results of Quasi-KCPCs are recorded in UF, while in Line 4 of Algorithm 4, UF is in initial state. The remaining parts of the algorithm are the same as Algorithm 4.

Theorem 14. *Algorithm 5 correctly finds all* KCPC*s in G.*

Theorem 15. *Let V_q be the set of vertices shared by any two* Quasi-KCPC*s, the time and space complexity of Algorithm 5 is $O(T_{clique} + T_{qkc} + m_q n_{clique}(\frac{\delta_q}{2})^{k-3})$ and $O(n_{clique}\delta)$ respectively, where T_{clique} is the time used in* MCE *algorithm, T_{qkc} is the time complexity of Algorithm 2, and m_q, δ_q are number of edges and degeneracy of subgraph induced by V_q.*

The space complexity of Algorithm 5 is $O(n_{clique}\delta)$ because the space complexity of the MCE tree is $O(n_{clique}\delta)$, as mentioned in the proof of Theorem 10.

6 Dynamic Maintenance of KCPC

Real-world networks can evolve through edge/vertex updates, requiring efficient dynamic maintenance of KCPC. In this section, we present practical algorithms for dynamic maintenance of KCPC under edge additions or deletions, vertex additions or deletions.

Edge addition. For edge addition, dynamic maintenance of KCPC requires considering two key aspects: (1) adding an edge may create new maximal cliques and invalidate existing ones, and (2) the connectivity between maximal cliques may also change, thereby altering the KCPC results. For newly formed maximal cliques, computation can be localized to the subgraph affected by the edge addition. For the remaining aspects, we employ an integrated approach that simultaneously updates KCPC while detecting invalidated maximal cliques (previously maximal). Specifically, updating KCPC requires traversing all adjacent maximal cliques of newly formed maximal cliques and counting their common vertices. On the other hand, any invalidated maximal clique must be adjacent to a new maximal clique, and the size of the invalidated clique equals the number of common vertices with the new maximal clique. Thus, these two processes can be integrated.

Algorithm 6 presents the process of updating KCPC after adding an edge. In Lines 1-4, the algorithm adds the new edge to graph G and computes the newly formed maximal cliques. In Lines 5-8, the algorithm traverses the adjacent maximal clique C of the newly

Algorithm 6: KCPC-edge addition

```
Input: A graph $G = (V, E)$; Integer $k$; Input edge $e = (u, v), u, v \in V$; Current
       maximal cliques $\mathcal{C}$; Current KCPC results $UF$
Output: Updated KCPCs
1  $G \leftarrow (V, E \cup \{(u, v)\})$;
2  $V' \leftarrow N_G(u) \cap N_G(v) \cup \{u, v\}$;
3  $G' \leftarrow$ induced subgraph of $G$ on $V'$;
4  $\mathcal{C}' \leftarrow$ maximal cliques in $G'$ with size at least $k$; $\mathcal{C} \leftarrow \mathcal{C} \cup \mathcal{C}'$;
5  for $C' \in \mathcal{C}'$ do
6      for $C \in \mathcal{C}$ which shares vertices with $C'$ do
7          if $|C \cap C'| \geq k - 1$ then $UF.union(C, C')$ ;
8          if $C \subseteq C'$ then Mark $C$ as invalidated maximal clique (previously maximal) ;
9  return $UF$;
```

formed maximal clique C'. If C, C' share at least $k-1$ vertices, they belong to the same KCPC (Line 7). If C is entirely contained by the new clique C', then C is no longer maximal (Line 8).

Theorem 16. *Algorithm 6 correctly updates* KCPC *after the addition of new edge.*

Proof. In Lines 5–8, the algorithm traverses all adjacent maximal cliques of the new ones to update KCPC. Moreover, in Line 8, since any invalidated maximal clique is guaranteed to be contained within a new maximal clique, it does not affect the correctness of KCPC update. Hence, it only needs to be marked, not deleted. As a result, the algorithm correctly updates KCPC. □

Theorem 17. *The time and space complexity of Algorithm 6 are $O(T'_{clique} + n'_{clique}(n'_{clique} + n_{clique})\delta)$ and $O((n_{clique} + n'_{clique})\delta)$, respectively, where T'_{clique} is the time to enumerate maximal cliques in G', n_{clique}, n'_{clique} are the number of current maximal cliques in $\mathcal{C}$ and new maximal cliques in $\mathcal{C}'$, respectively. δ is the degeneracy of G.*

Proof. Algorithm 6 consists only of two main steps: enumerating maximal cliques on the induced subgraph G', and traversing the adjacent maximal cliques of the newly generated ones. Thus, the time complexity is $O(T'_{clique}+n'_{clique}(n'_{clique}+n_{clique})\delta)$. Moreover, in Algorithm 6, the current and newly generated maximal cliques occupy the majority of the space. As a result, the space complexity is $O((n_{clique} + n'_{clique})\delta)$. □

Edge deletion. For edge deletion, dynamic maintenance of KCPC requires considering two similar key aspects: (1) deleting an edge leads to the removal of the maximal cliques containing it, and several smaller new maximal cliques may be generated; (2) the connectivity between maximal cliques may change, thereby altering the KCPC results. Unlike the case of edge addition, edge deletion requires the complete removal of the affected maximal cliques, which may cause the original KCPC to split into multiple separate KCPCs. Meanwhile, the newly generated smaller maximal cliques must be subsets of the deleted maximal cliques.

Algorithm 7 presents the process of updating KCPC after deleting an edge. After deleting e, all maximal cliques containing e are removed (Lines 2–3). In Lines 4–5, the algorithm updates only the affected KCPCs —those containing the removed cliques. Then, in Lines 7–10, for each removed clique, two new candidate maximal cliques are generated by removing u and v, respectively. A candidate is a new maximal clique only if it is not contained in any adjacent maximal clique. Finally, in Lines 11–14, the algorithm updates the KCPCs with the new maximal cliques.

Theorem 18. *Algorithm 7 correctly updates* KCPC *after the deletion of an edge.*

Algorithm 7: KCPC-edge deletion

```
Input: A graph $G = (V, E)$; Integer $k$; Input edge $e = (u, v), e \in E$; Current maximal
       cliques $\mathcal{C}$; Current KCPC results $UF$
Output: Updated KCPCs
1  $G \leftarrow (V, E - \{(u, v)\})$;
2  $\mathcal{C}' \leftarrow$ set of maximal cliques in $\mathcal{C}$ containing $e$;
3  $\mathcal{C} \leftarrow \mathcal{C} - \mathcal{C}'$;
4  $\hat{\mathcal{C}} \leftarrow \{C | C \in \mathcal{C}, \exists C' \in \mathcal{C}', UF.find(C) = UF.find(C')\}$;
5  Recompute KCPC ($UF$) on the subgraph induced by maximal cliques in $\hat{\mathcal{C}}$;
6  $\mathcal{NC} \leftarrow \emptyset$;
7  for $C' \in \mathcal{C}'$ do
8      $C_1 \leftarrow C' - \{u\}$; $C_2 \leftarrow C' - \{v\}$;
9      if $|C_1| \geq k$ and $\nexists C \in \mathcal{C}, C_1 \subseteq C$ then $\mathcal{NC} \leftarrow \mathcal{NC} \cup \{C_1\}$;
10     if $|C_2| \geq k$ and $\nexists C \in \mathcal{C}, C_2 \subseteq C$ then $\mathcal{NC} \leftarrow \mathcal{NC} \cup \{C_2\}$;
11 for $NC \in \mathcal{NC}$ do
12     for $C \in \mathcal{C}$ which shares vertices with $NC$ do
13         if $|NC \cap C| \geq k - 1$ then $UF.union(C, NC)$;
14         $\mathcal{C} \leftarrow \mathcal{C} \cup \{NC\}$;
15 return $UF$;
```

Proof. In Lines 4–5, the algorithm updates only the KCPCs related to the deleted maximal cliques, since other KCPCs remain unaffected and do not need to be updated. In Lines 11–14, it traverses all adjacent maximal cliques of each new maximal clique to correctly update the KCPCs. □

Theorem 19. *The time and space complexity of Algorithm 7 are $O(\hat{T} + n'_{clique}(n'_{clique} + n_{clique})\delta)$ and $O((n_{clique} + n'_{clique})\delta)$ respectively, where $\hat{T}$ is the time for recomputing* KCPC *in Line 5, n_{clique}, n'_{clique} are the number of current maximal cliques in $\mathcal{C}$ and deleted maximal cliques in $\mathcal{C}'$, respectively. δ is the degeneracy of G.*

Proof. Note that the size of $\mathcal{NC}$ is at most $2n'_{clique}$. The rest of the proof follows similarly to that of Theorem 17. □

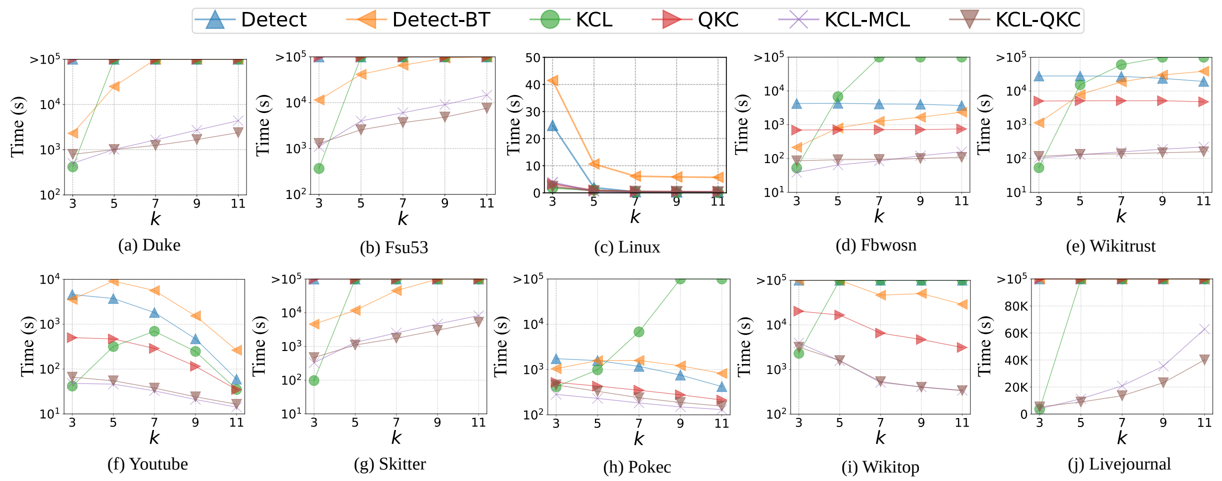
Avoid recomputation in edge deletion. Algorithm 7 recomputes KCPC in Line 5. However, in real-world large graphs, the adjacency graph of maximal cliques tends to be dense (see Table 2: N_m, N_o). As a result, deleting a small number of maximal cliques rarely causes the KCPC to split. Therefore, local recomputation in Line 5 is often unnecessary. In practice, we can simply check the connectivity among the adjacent maximal cliques of the deleted cliques—if they remain connected, KCPC recomputation can be skipped.

Vertex addition and deletion. The dynamic maintenance for adding and deleting vertices is very similar to that for edges. The maintenance algorithm for vertex addition can be adapted from Algorithm 6. Specifically, Lines 1–4 can be modified to add the new vertex and its associated edges to the graph, followed by computing the maximal cliques that include the new vertex. The maintenance algorithm for vertex deletion can be adapted from Algorithm 7. Specifically, Lines 1–2 can be modified to delete the vertex and set $\mathcal{C}'$ as the set of maximal cliques containing the deleted vertex. In Lines 7–10, the new maximal cliques in $\mathcal{NC}$ are derived from the cliques in $\mathcal{C}'$ by removing the deleted vertex.

7 Experiments

7.1 Experimental Setup

Datasets. We use 12 real-world datasets in our experiments, as detailed in Table 2. Among them, Wikitop is downloaded from SNAP (http://snap.stanford.edu/index.html), and all other datasets are downloaded from (https://networkrepository.com/). d_{avg} denotes the average degree of vertices in each datasets. ω represents the size of the maximum clique. N_m is the number of



(a) Duke (b) Fsu53 (c) Linux (d) Fbwosn (e) Wikitrust (f) Youtube (g) Skitter (h) Pokec (i) Wikitop (j) Livejournal

Figure 7: Performance of different algorithms on different datasets, varying k

Table 2: Datasets, d_{avg} is the average degree of vertices, ω is the size of the maximum clique, N_m is the number of maximal cliques, N_o is the number of edges in adjacent graph of maximal cliques, K=10^3, M=10^6, B=10^9, T=10^{12}, *TO*='timeout'

Datasets	$\|V\|$	$\|E\|$	d_{avg}	ω	N_m	N_o
Duke	9,885	506,437	102.47	34	8.5M	19T
Fsu53	27,737	1,034,802	74.62	56	11M	6T
Linux	30,837	213,217	13.83	10	111K	168M
Fbwosn	63,731	817,090	25.64	30	1.4M	29B
Wikitrust	138,587	715,883	10.33	25	1.2M	277B
Youtube	1,157,827	2,987,624	5.16	17	1.6M	75B
Skitter	1,218,581	3,341,589	5.48	64	6.4M	*TO*
Pokec	1,632,803	22,301,964	27.32	29	12M	20B
Wikitop	1,791,488	25,444,207	28.4	39	23M	2T
Livejournal	4,033,137	27,933,062	13.85	214	28M	*TO*
Uci-uni	58,790,782	92,208,194	3.13	20	6.4M	48B
Konect	59,216,214	92,522,011	3.12	26	6.5M	11B

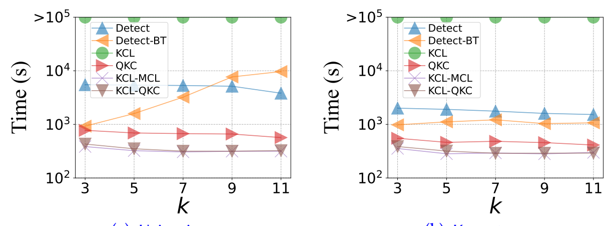
maximal cliques larger than 2. N_o is the number of edges in the maximal clique adjacency graph. As shown in Table 2, N_o is often very large, indicating that existing MCE-based solutions are often inefficient when applied to real-world graphs.

Algorithms. We implement three state-of-the-art algorithms for comparison, including two MCE-based solutions and one KCL-based solution. The details of these algorithms are as follows:

- Detect [60] is an MCE-based algorithm inspired by the approach proposed in [47]. The key difference lies in the use of adjacency lists to construct the clique adjacency graph (MCAG), whereas [47] employs adjacency matrices.
- Detect-BT [51] is another MCE-based algorithm based on the idea from [47]. Like Detect, it uses adjacency lists to build the MCAG but introduces a binary tree structure to reorganize the maximal cliques, thereby reducing the cost of traversing the MCAG.
- KCL [5] is the state-of-the-art KCL-based algorithm. It is based on listing all k-cliques and $(k-1)$-cliques, and connects k-cliques sharing the same $(k-1)$-clique.

We implement three proposed algorithms (both maximal clique enumeration and MCE tree used for computing Quasi-KCPC are based on [13, 14]):

- QKC is our MCE-based solution (Algorithm 3). In QKC, the MCE algorithm is used to compute all maximal cliques and construct the MCE tree. Algorithm 3 then invokes Algorithm 2 to generate Quasi-KCPCs. Subsequently, it traverses the smaller-scale MCAG based on these Quasi-KCPCs to obtain the final KCPCs.
- KCL-MCL is our KCL-based algorithm (Algorithm 4). In this approach, maximal cliques are utilized to prune the search space for k-clique listing.
- KCL-QKC is another KCL-based algorithm (Algorithm 5). Here, Quasi-KCPCs are employed to prune the search space for k-clique listing.



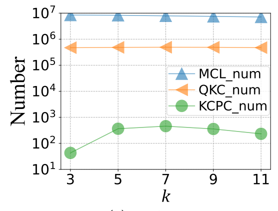
(a) Uci-uni (b) Konect

Figure 8: Performance of different algorithms on Uci-uni and Konect

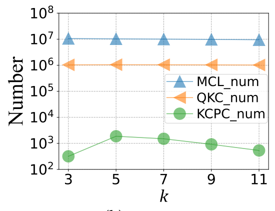
We also implemented four KCPC dynamic update algorithms: Add-edge (Algorithm 6), Del-edge (Algorithm 7), Add-vertex and Del-vertex. These four algorithms correspond to the dynamic KCPC updates for adding an edge, deleting an edge, adding a vertex, and deleting a vertex, respectively.

Parameters. There are two parameters in the experiments, k and l. l is used for QKC and KCL-QKC. During the execution of the MCE algorithm within these two algorithms, P is the candidate set of each search tree node. If $|P| \leq l$, then the pivot technique is not applied. This will result in more search tree branches under the corresponding search tree node, leading to repeated enumerations. However, it will also produce more p-prefixes ($p \geq k$), allowing more maximal cliques to be connected to Quasi-KCPC. The larger the value of l, the more search tree nodes do not apply the pivot technique. In our experiments, the default value of l is set to 10, as our algorithms perform very good under this setting.

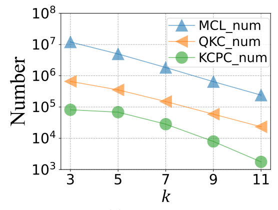
All the algorithms are implemented in C++. The experiments are conducted on a system running Ubuntu 20.04.4 LTS with an AMD Ryzen 3990X 2.2GHz CPU and 128GB of memory.



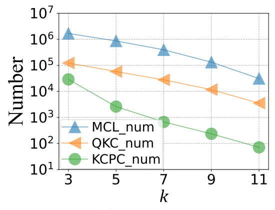
(a) Duke



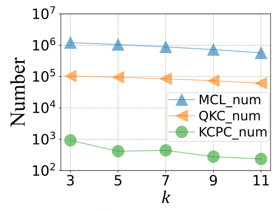
(b) Fsu53



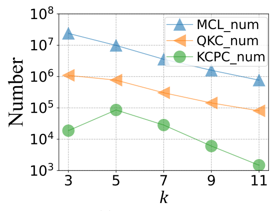
(c) Pokec



(d) Youtube



(e) Wikitrust



(f) Wikitop

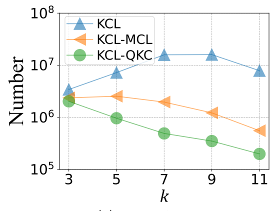
Figure 9: The number of maximal cliques (≥ k, MCL_num), Quasi-KCPCs (QKC_num) and KCPCs (KCPC_num) in QKC and KCL-QKC, varying k

7.2 Experimental Results

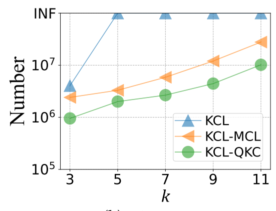
Exp-1: Efficiency of different algorithms. In this experiment, we evaluate the efficiency of various algorithms on all datasets. Figure 7 and 8 show the results.

Efficiency of existing algorithms. Among existing algorithms, Detect performs the worst overall due to its need to traverse the entire MCAG. While Detect-BT generally outperforms Detect, it still struggles on datasets like Youtube and Linux. KCL is efficient when k is small, but becomes impractical as k increases due to the exponential growth of the number of k-cliques. On Uci-uni and Konect, KCL even runs out of memory when $k = 3$. In contrast, our algorithm significantly outperforms all baselines. For example, on the Wikitrust dataset with $k = 7$, KCL-QKC completes in 138 seconds, compared to 27,000 seconds for Detect, 18,000 seconds for Detect-BT, and over 60,000 seconds for KCL.

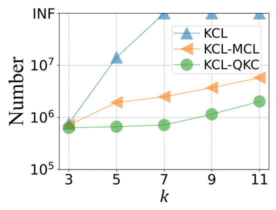
Efficiency of our methods. QKC outperforms all existing methods on most datasets, except for Duke, Fsu53, Skitter, and Livejournal, which are globally or locally dense (with large d_{avg} or ω). These graphs yield exceptionally large MCAG —even after Quasi-KCPC based reduction (see Table 2 and Figure 9). In contrast, KCL-MCL and KCL-QKC outperform all baselines by avoiding full MCAG traversal and effectively pruning the search space. While both perform similarly on sparse graphs, KCL-QKC shows clear advantages on dense graphs. For instance, on Fsu53 with $k = 11$, KCL-QKC takes 7,500 seconds versus 14,000 for KCL-MCL. This is because dense graphs contain a vast number of k-cliques as k increases, and Quasi-KCPC in KCL-QKC offers stronger pruning, resulting in better efficiency.



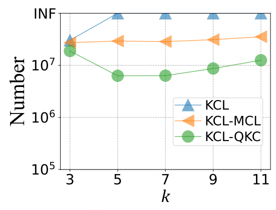
(a) Youtube



(b) Skitter



(c) Wikitrust



(d) Livejournal

Figure 10: The number of search tree nodes of k-clique listing in KCL, KCL-MCL and KCL-QKC, varying k

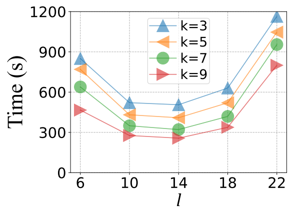
Table 3: Number of Quasi-KCPCs with varying l

	$l = 6$	$l = 7$	$l = 8$	$l = 9$	$l = 10$	$l = 11$	$l = 12$	$l = 13$	$l = 14$	$l = 15$
Pokec	1,106,376	931,931	814,096	730,896	668,988	620,852	581,689	547,990	518,452	492,096
Fbwosn	261,245	189,927	137,320	101,808	76,011	57,924	45,232	35,852	29,221	24,220
Youtube	238,864	203,096	177,136	157,612	142,698	130,855	120,788	112,632	106,032	100,023
Linux	9,295	7,757	6,630	5,841	5,130	4,240	3,694	3,270	2,913	2,604
	$l = 16$	$l = 17$	$l = 18$	$l = 19$	$l = 20$	$l = 21$	$l = 22$	$l = 23$	$l = 24$	$l = 25$
Pokec	467,493	444,640	422,707	401,533	381,145	361,287	342,476	323,472	304,948	286,337
Fbwosn	20,317	17,548	15,304	13,532	12,201	11,022	10,033	9,256	8,496	7,911
Youtube	94,946	90,345	86,347	82,824	79,417	76,330	73,777	71,557	69,135	66,911
Linux	2,192	2,016	1,732	1,373	1,108	893	717	514	514	514

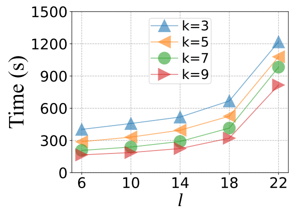
The trend of the curves. As shown in Figure 7, the performance curves vary with k due to multiple influencing factors. One is the number of maximal cliques with at least k vertices, which decreases as k increases, leading to lower runtime for maximal clique based methods (Detect, QKC), as seen in Youtube. Another factor is the number of k-cliques, which typically increases then decreases with k, with dataset-dependent inflection points. Consequently, KCL generally shows increasing trends, though in some cases like Youtube, it first rises then falls. Since KCL-MCL and KCL-QKC are influenced by both factors, their trends are more complex and less predictable.

Exp-2: The effectiveness of Quasi-KCPC in QKC and KCL-QKC. This experiment evaluates the effectiveness of Quasi-KCPC in QKC and KCL-QKC by comparing the number of maximal cliques of size at least k. As shown in Figure 9, across all six subfigures, the number of Quasi-KCPCs is consistently about an order of magnitude smaller than the number of maximal cliques, and closer to the number of KCPCs. This reduction allows QKC to traverse a much smaller MCAG and enables KCL-QKC to perform more effective pruning, highlighting the efficiency of our proposed methods.

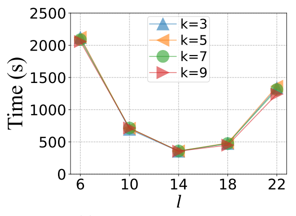
Exp-3: The pruning effect of maximal cliques and Quasi-KCPCs in KCL-MCL and KCL-QKC. This experiment evaluates the pruning effect of maximal cliques and Quasi-KCPCs in KCL-MCL and KCL-QKC. Figure 10 shows the number of search tree nodes generated during k-clique listing. Across all datasets, KCL produces significantly more nodes than KCL-MCL and KCL-QKC, confirming the pruning power of both maximal cliques and Quasi-KCPCs. Notably, KCL-QKC generates the fewest nodes, indicating that Quasi-KCPCs offer stronger pruning than maximal cliques.



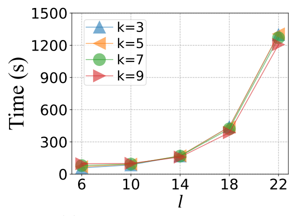
(a) QKC on Pokec



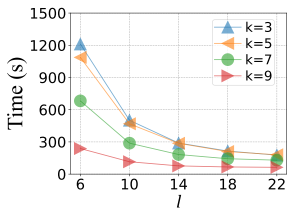
(b) KCL-QKC on Pokec



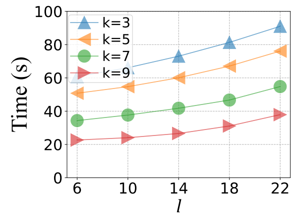
(c) QKC on Fbwosn



(d) KCL-QKC on Fbwosn



(e) QKC on Youtube



(f) KCL-QKC on Youtube

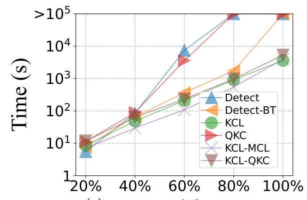
Figure 11: The impact of pivot selection in QKC and KCL-QKC, varying l

Exp-4: The impact of pivot selection (varying l). In this experiment, we evaluate the impact of pivot selection by varying l in QKC and KCL-QKC. We first evaluate how the parameter l affects the approximation between Quasi-KCPC and KCPC. As shown in Table 3, the number of Quasi-KCPCs consistently decreases as l increases, indicating closer approximation to KCPCs and confirming Theorem 8. On Linux, when $l \geq 23$ (its degeneracy), the number of Quasi-KCPCs equals that of KCPCs (514), verifying Theorem 7. However, larger l also increases computation overhead of Quasi-KCPC. Figure 11 shows the impact of varying l on QKC and KCL-QKC across Pokec, Fbwosn, and Youtube. For QKC, in Pokec and Fbwosn, runtime first decreases then increases, while in Youtube, it keeps decreasing. This is because Youtube is sparser, making Quasi-KCPC computation less costly even as l increases. In denser graphs like Pokec and Fbwosn, larger l increases computation overhead of Quasi-KCPC, causing earlier performance degradation. For KCL-QKC, increasing l consistently degrades performance across all datasets, as precomputed Quasi-KCPCs already provide strong pruning, and the added computation becomes non-negligible. Overall, in QKC, how the efficiency of QKC changes with the degree of pivot selection usage is more dependent on the dataset itself, whereas in KCL-QKC, maintaining sufficient pivot selection usage (keeping small l) leads to high efficiency.

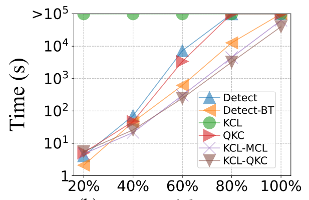
Exp-5: The effect of color ordering. In this experiment, we evaluate the effect of color ordering (Col) in the progress of k-clique listing of KCL-QKC. We use degeneracy ordering (Degen) for comparison. Table 4 shows the results on four datasets. The results on the other datasets are consistent. We can observe that Col slightly outperforms Degen in most cases. These results, combined with the performance of KCL, suggest that the superior performance of KCL-QKC should be primarily attributed to the pruning effect of Quasi-KCPCs, since the performance of KCL (without Quasi-KCPC pruning) is poor, and both Col and Degen perform well, while Col does not show a significant improvement compared to Degen.

Table 4: Performance (seconds) of KCL-QKC with color ordering and degeneracy ordering

	Pokec		Youtube		Wikitrust		Wikitop	
	Col	Degen	Col	Degen	Col	Degen	Col	Degen
$k = 3$	**458**	487	**66**	120	**116**	170	**3168**	8033
$k = 5$	**331**	341	**55**	77	**133**	168	**1587**	2707
$k = 7$	**239**	244	**38**	42	**138**	152	**537**	624
$k = 9$	**187**	188	24	24	149	**135**	401	**397**
$k = 11$	**156**	159	**16**	17	159	**122**	337	**324**



(a) Livejounal, $k = 3$



(b) Livejounal, $k = 11$

Figure 12: Scalability of all algorithms

Table 5: Memory usage (MB) of the algorithms, $k = 3$

	Detect	Detect-BT	KCL	QKC	KCL-MCL	KCL-QKC
Youtube	719	1,370	5,511	842	1,376	1,014
Skitter	1,600	10,441	6,345	2,483	3,684	2,505
Pokec	3,466	8,642	12,835	4,364	7,683	6,216
Wikitop	4,172	15,853	17,471	5,884	12,622	7,550
Livejournal	13,918	121,040	31,246	21,429	25,922	20,070
Uci-uni	26,238	16,568	>169,630	26,520	32,029	28,009
Konect	26,319	16,731	>151,714	26,318	32,099	27,927

Exp-6: Scalability. We evaluate scalability using dataset Livejournal by sampling 20%, 40%, 60%, and 80% of vertices and running experiments on the induced subgraphs. As shown in Figure 12, KCL-MCL and KCL-QKC show the best scalability, particularly at $k = 3$. KCL also performs well at small k due to fewer k-cliques. While Detect-BT performs reasonably on sampled graphs, it fails on the full graph due to excessive memory usage. These results demonstrate the scalability of our proposed methods on large graphs.

Exp-7: Memory overhead. Tables 5 and 6 report memory usage for all algorithms on 7 large datasets with $k = 3$ and $k = 11$. KCL consumes the most memory, while Detect or Detect-BT uses the least. For most algorithms (except KCL), memory usage tends to be lower at $k = 11$ due to fewer maximal cliques. In contrast, memory of KCL grows with the number of k-cliques and continues increasing until termination. KCL-MCL and KCL-QKC use slightly more memory than Detect but deliver significantly better performance, demonstrating their high memory efficiency.

Exp-8: Dynamic maintenance of KCPC. This experiment evaluates the average per-update time (ms) of KCPC update algorithms, with results shown in Tables 7 and 8 for Skitter and Livejournal, respectively. We process 10,000 edges (or vertices) in 5 batches of 2,000 each. For each batch, the reported time is the average time per addition or deletion; "re-run" indicates the average time of full recomputation using KCL-QKC. It can be observed that all update algorithms are significantly faster than recomputation. For instance,

Table 6: Memory usage (MB) of the algorithms, $k = 11$

	Detect	Detect-BT	KCL	QKC	KCL-MCL	KCL-QKC
Youtube	696	387	5,419	696	696	696
Skitter	1,501	9,631	>13,817	2,269	1,985	2,026
Pokec	3,646	2,179	>31,464	3,646	3,646	3,646
Wikitop	3,616	3,171	>79,940	3,617	4,127	4,127
Livejournal	12,850	99,201	>49,961	18,938	13,866	15,091
Uci-uni	26,238	13,226	>141,536	26,238	27,846	28,092
Konect	26,319	13,411	>155,432	26,318	29,075	28,158

Table 7: Average time (ms) of KCPC dynamic maintenance on Skitter

	batch 1	batch 2	batch 3	batch 4	batch 5	re-run
Add-edge	0.84	0.91	0.81	0.83	0.83	≥467601
Del-edge	106.00	106.38	106.38	106.58	103.76	≥472548
Add-vertex	1.04	0.97	0.97	0.99	0.59	≥469794
Del-vertex	103.40	103.7	103.70	103.42	103.41	≥478494

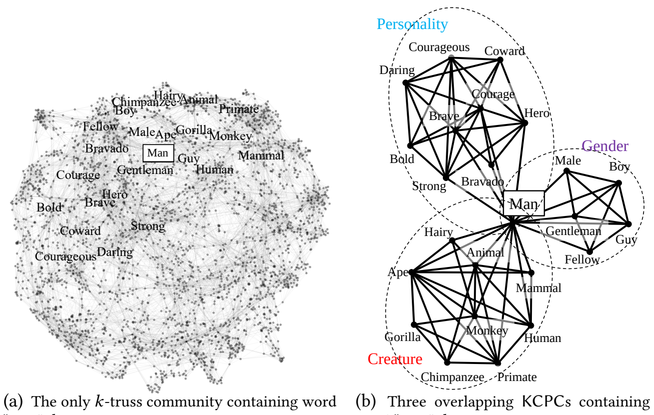
Table 8: Average time (ms) of KCPC dynamic maintenance on Livejournal

	batch 1	batch 2	batch 3	batch 4	batch 5	re-run
Add-edge	5.59	4.97	23.79	22.58	4.59	≥5273307
Del-edge	6166.49	5760.93	1630.75	658.23	5755.9	≥5315694
Add-vertex	7.57	6.61	6.55	5.71	5.10	≥5332485
Del-vertex	531.07	748.90	57848.52	6169.5	527	≥5371755

on Livejournal, Del-vertex takes 57s on average in batch 3, while re-running KCL-QKC takes at least 5371s. Overall, deletions are more expensive than additions due to their greater impact on KCPC connectivity.

Exp-9: Case study. We conduct a case study on a word association network WordNet [28]. In this network, a vertex represents a word, and an edge between two words indicates that they are meaningfully related. We use k-truss community [29] as a comparison with KCPC. Similar to KCPC, k-truss community can also be utilized as an overlapping community model [29]. It is a maximal set of edges which are connected by series of triangles, and each edge is contained in at least $k-2$ triangles (see [29]). For example, two triangles sharing a vertex are two overlapping 3-truss communities. Let $k = 5$, we present all k-truss communities and KCPCs containing word "Man" in WordNet (Figure 13). We find that there are three overlapping KCPCs in Figure 13 (b), representing three different word association communities which can be used to describe *Creature*, *Personality* and *Gender* respectively. All these communities share word "Man", because "Man" indeed has three different meanings above. However, in Figure 13 (a), we only find one large k-truss community containing "Man" and all the other related words in Figure 13 (b). The potential reason is that k-truss is less cohesive than k-clique. This result indicates that KCPC can model overlapping communities more accurately compared to the k-truss community model.

Summary and Recommendation. In terms of runtime, KCL-MCL and KCL-QKC outperform all baselines, with KCL-QKC showing better results on dense graphs. Detect-BT improves over Detect by optimizing MCAG traversal, while QKC, leveraging Quasi-KCPC, generally outperforms both. KCL performs well for small k, but degrades quickly as k increases. For memory usage, Detect is the most efficient due to its simplicity. QKC, KCL-MCL, and KCL-QKC also maintain low overhead. In contrast, Detect-BT exhibits unstable performance owing to data structures used for traversal acceleration. KCL consumes the most memory due to the need to store large number of k-cliques.



(a) The only k-truss community containing word "Man", $k = 5$ (b) Three overlapping KCPCs containing word "Man", $k = 5$

Figure 13: Case study on WordNet

Table 9: Summary of the performances of all algorithms for KCPC mining (the more stars, the better. ☆ denotes an uncertain star ranging from 0-1)

Algorithm	Small k	Large k	Memory	Overall
Detect	★	★★	★★★★★	★★
Detect-BT	★★★	★☆	★☆☆	★★
KCL	★★★★☆	☆	☆	★☆
QKC	★★	★★★	★★★★	★★★
KCL-MCL	★★★★★	★★★★☆	★★★★	★★★★☆
KCL-QKC	★★★★★	★★★★★	★★★★	★★★★★

We have summarized our recommendations for each algorithm in Table 9, where more stars indicate a higher recommendation. Overall, for practical applications, KCL-QKC stands out as the top choice for KCPC mining tasks, offering a balance of high performance and memory efficiency. KCL-MCL is a strong alternative, especially for scenarios where ease of development is a priority. While Detect and Detect-BT have their merits, they are outperformed by our more advanced algorithms in most cases. KCL is suitable only for small k values and is less practical for larger-scale problems.

8 Related Work

Community detection (see [19] as an entrance) plays a pivotal role in network analysis, with a main focus on disjoint communities, which has been conducted via modularity-based optimization, random walk, and information theory [3, 18, 27, 34, 44, 45, 63]. However, an object can participate in multiple communities, overlapping community detection has gained substantial attention. Overlapping communities can be defined using different methodologies [15, 29, 41, 47, 49, 62], including the communities defined as cliques [47, 62], sets of edges [15, 49], trusses [29] and local expansion starting from particular vertices [41].

Within overlapping community detection, the k-clique percolation community (KCPC) detection has been extensively studied [2, 5, 21, 22, 33, 47, 51]. It presents two types of solutions: k-clique based solutions [5, 33] and maximal clique-based solutions [2, 47, 51]. The algorithms in [5, 33] offer a complexity linear to the number of k-cliques in the network. Alternatively, [2, 47] employ adjacency matrices to construct the maximal clique graph and extract the KCPC by identifying connected components within this matrix. [51] utilizes adjacency lists and uses BFS traversal with pruning techniques. Parallel approaches [21, 22] are also proposed to extract the KCPC on the adjacency graph of maximal cliques. Existing

KCPC methods are primarily suited for small-scale graphs. There is still a need for scalable algorithms for KCPC detection due to its high complexity. Additionally, many researchers turned to study various variants of KCPC [10, 17, 36, 37, 48], including KCPC in weighted graphs [17], directed graphs [48], random graphs [10, 37] and bipartite graphs [36].

Our work is also related to the maximal clique enumeration and k-clique listing. The maximal clique enumeration is a fundamental tool in clique-based community detection for which the Bron-Kerbosch algorithm and its variants [6, 14, 43, 55] are often used. For k-clique listing, various approaches have been explored [9, 32, 39, 61], including techniques such as vertex ordering [9, 39, 61] and pivoting [32].

9 Conclusion

In this paper, we address the problem of k-clique percolation community (KCPC) mining, which is a fundamental operator in graph analysis applications. We propose several novel and efficient algorithms from both maximal clique enumeration and k-clique listing perspectives. Specifically, from the perspective of maximal clique enumeration, we introduce a new concept called Quasi-KCPC, a compact intermediate structure that can be efficiently extracted during the enumeration process. By leveraging Quasi-KCPC, we significantly reduce the size of the maximal clique adjacency graph and improve the efficiency of KCPC mining. From the perspective of k-clique listing, we propose two algorithms that connect maximal cliques through shared $(k-1)$-cliques, in contrast to traditional methods that connect k-cliques directly. This design enables substantial pruning of the search space by exploiting the containment hierarchy between cliques and their associated maximal cliques or Quasi-KCPCs. We also propose efficient update algorithms for dynamic KCPC updates. These algorithms adopt local update strategies and integrate the maintenance of maximal cliques and KCPCs. We conduct comprehensive experiments to evaluate the efficiency and effectiveness of our algorithms using 12 real-world graphs, and the results demonstrate the superiority of our proposed solutions.

References

[1] Zahra Ebadi Abouzar, Leila Esmaeili, and Alireza Hashemi Golpayegani. 2016. Centrality measures analysis in overlapped communities: An empirical study. In *2016 8th International Symposium on Telecommunications (IST)*. IEEE, 565–570.

[2] Balázs Adamcsek, Gergely Palla, Illés J Farkas, Imre Derényi, and Tamás Vicsek. 2006. CFinder: locating cliques and overlapping modules in biological networks. *Bioinformatics* 22, 8 (2006), 1021–1023.

[3] Jess Banks, Cristopher Moore, Joe Neeman, and Praneeth Netrapalli. 2016. Information-theoretic thresholds for community detection in sparse networks. In *Conference on Learning Theory*. PMLR, 383–416.

[4] Vladimir Batagelj and Matjaz Zaversnik. 2003. An o (m) algorithm for cores decomposition of networks. *arXiv preprint cs/0310049* (2003).

[5] Alexis Baudin, Maximilien Danisch, Sergey Kirgizov, Clémence Magnien, and Marwan Ghanem. 2022. Clique percolation method: memory efficient almost exact communities. In *International Conference on Advanced Data Mining and Applications*. Springer, 113–127.

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

[7] Lijun Chang and Lu Qin. 2019. Cohesive subgraph computation over large sparse graphs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 2068–2071.

[8] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to algorithms*. MIT press.

[9] 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.

[10] Imre Derényi, Gergely Palla, and Tamás Vicsek. 2005. Clique percolation in random networks. *Physical review letters* 94, 16 (2005), 160202.

[11] Jia-Qi Dong, Zhou Shen, Yongwen Zhang, Zi-Gang Huang, Liang Huang, and Xiaosong Chen. 2018. Finite-size scaling of clique percolation on two-dimensional Moore lattices. *Physical Review E* 97, 5 (2018), 052133.

[12] Yon Dourisboure, Filippo Geraci, and Marco Pellegrini. 2007. Extraction and classification of dense communities in the web. In *Proceedings of the 16th international conference on World Wide Web*. 461–470.

[13] David Eppstein, Maarten Löffler, and Darren Strash. 2010. Listing all maximal cliques in sparse graphs in near-optimal time. In *Algorithms and Computation: 21st International Symposium, ISAAC 2010, Jeju Island, Korea, December 15-17, 2010, Proceedings, Part I 21*. Springer, 403–414.

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

[15] Tim S Evans and Renaud Lambiotte. 2009. Line graphs, link partitions, and overlapping communities. *Physical review E* 80, 1 (2009), 016105.

[16] Jingfang Fan and Xiaosong Chen. 2014. General clique percolation in random networks. *Europhysics Letters* 107, 2 (2014), 28005.

[17] Illés Farkas, Dániel Ábel, Gergely Palla, and Tamás Vicsek. 2007. Weighted network modules. *New Journal of Physics* 9, 6 (2007), 180.

[18] Md Abdul Motaleb Faysal and Shaikh Arifuzzaman. 2019. Distributed community detection in large networks using an information-theoretic approach. In *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 4773–4782.

[19] Santo Fortunato. 2010. Community detection in graphs. *Physics reports* 486, 3-5 (2010), 75–174.

[20] Michelle Girvan and Mark EJ Newman. 2002. Community structure in social and biological networks. *Proceedings of the national academy of sciences* 99, 12 (2002), 7821–7826.

[21] Enrico Gregori, Luciano Lenzini, and Simone Mainardi. 2012. Parallel k-clique community detection on large-scale networks. *IEEE Transactions on Parallel and Distributed Systems* 24, 8 (2012), 1651–1660.

[22] Enrico Gregori, Luciano Lenzini, Simone Mainardi, and Chiara Orsini. 2012. FLIP-CPM: A Parallel Community Detection Method. In *Computer and Information Sciences II: 26th International Symposium on Computer and Information Sciences*. Springer, 249–255.

[23] Enrico Gregori, Luciano Lenzini, and Chiara Orsini. 2011. k-clique Communities in the Internet AS-level Topology Graph. In *2011 31st International Conference on Distributed Computing Systems Workshops*. IEEE, 134–139.

[24] Sumit Kumar Gupta, Dhirendra Pratap Singh, and Jaytrilok Choudhary. 2022. A review of clique-based overlapping community detection algorithms. *Knowledge and Information Systems* 64, 8 (2022), 2023–2058.

[25] Xiaoxu Han, Cheng Li, Shichao Sun, Jiaojiao Ji, Bao Nie, Garth Maker, Yonglin Ren, and Li Wang. 2022. The chromosome-level genome of female ginseng (Angelica sinensis) provides insights into molecular mechanisms and evolution of coumarin biosynthesis. *The Plant Journal* 112, 5 (2022), 1224–1237.

[26] Lu Huang, Fangyan Liu, and Yi Zhang. 2020. Overlapping community discovery for identifying key research themes. *IEEE transactions on engineering management* 68, 5 (2020), 1321–1333.

[27] Ling Huang, Chang-Dong Wang, and Hong-Yang Chao. 2018. A harmonic motif modularity approach for multi-layer network community detection. In *2018 IEEE International Conference on Data Mining (ICDM)*. IEEE, 1043–1048.

[28] Xin Huang, Hong Cheng, Rong-Hua Li, Lu Qin, and Jeffrey Xu Yu. 2015. Top-k structural diversity search in large networks. *The VLDB Journal* 24, 3 (2015), 319–343.

[29] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying k-truss community in large and dynamic graphs. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 1311–1322.

[30] Pan Hui and Jon Crowcroft. 2008. Human mobility models and opportunistic communications system design. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 366, 1872 (2008), 2005–2016.

[31] Sminu Izudheen, Eljose S Sajan, Ivan George, Jeevan John, and Chris Shaju Attipetty. 2020. Effect of community structures in protein–protein interaction network in cancer protein identification. *Current Science* 118, 1 (2020), 62–69.

[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] Jussi M Kumpula, Mikko Kivelä, Kimmo Kaski, and Jari Saramäki. 2008. Sequential algorithm for fast clique percolation. *Physical review E* 78, 2 (2008), 026109.

[34] Darong Lai, Hongtao Lu, and Christine Nardini. 2010. Enhanced modularity-based community detection by random walk network preprocessing. *Physical Review E* 81, 6 (2010), 066118.

[35] Chiyoung Lee, Jiepin Cao, and Rosa Gonzalez-Guarda. 2023. The application of clique percolation method in health disparities research. *Journal of Advanced Nursing* 79, 11 (2023), 4318–4325.

[36] Sune Lehmann, Martin Schwartz, and Lars Kai Hansen. 2008. Biclique communities. *Physical review E* 78, 1 (2008), 016108.

[37] Ming Li, Youjin Deng, and Bing-Hong Wang. 2015. Clique percolation in random graphs. *Physical Review E* 92, 4 (2015), 042116.

[38] Ming Li, Run-Ran Liu, Linyuan Lü, Mao-Bin Hu, Shuqi Xu, and Yi-Cheng Zhang. 2021. Percolation on complex networks: Theory and application. *Physics Reports* 907 (2021), 1–68.

[39] Ronghua Li, Sen Gao, Lu Qin, Guoren Wang, Weihua Yang, and Jeffrey Xu Yu. 2020. Ordering Heuristics for k-clique Listing. *Proc. VLDB Endow.* (2020).

[40] Rong-Hua Li, Qiushuo Song, Xiaokui Xiao, Lu Qin, Guoren Wang, Jeffrey Xu Yu, and Rui Mao. 2020. I/O-efficient algorithms for degeneracy computation on massive networks. *IEEE Transactions on Knowledge and Data Engineering* 34, 7 (2020), 3335–3348.

[41] Jian Ma and Jianping Fan. 2019. Local optimization for clique-based overlapping community detection in complex networks. *IEEE Access* 8 (2019), 5091–5103.

[42] Xuelian Ma, Hengyu Yan, Jiaotong Yang, Yue Liu, Zhongqiu Li, Minghao Sheng, Yaxin Cao, Xinyue Yu, Xin Yi, Wenying Xu, et al. 2022. PlantGSAD: a comprehensive gene set annotation database for plant species. *Nucleic Acids Research* 50, D1 (2022), D1456–D1467.

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

[44] Mark EJ Newman. 2006. Modularity and community structure in networks. *Proceedings of the national academy of sciences* 103, 23 (2006), 8577–8582.

[45] Makoto Okuda, Shin'ichi Satoh, Yoichi Sato, and Yutaka Kidawara. 2019. Community detection using restrained random-walk similarity. *IEEE transactions on pattern analysis and machine intelligence* 43, 1 (2019), 89–103.

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

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

[48] Gergely Palla, Illés J Farkas, Péter Pollner, Imre Derényi, and Tamás Vicsek. 2007. Directed network modules. *New journal of physics* 9, 6 (2007), 186.

[49] Alexander Ponomarenko, Leonidas Pitsoulis, and Marat Shamshetdinov. 2021. Overlapping community detection in networks based on link partitioning and partitioning around medoids. *Plos one* 16, 8 (2021), e0255717.

[50] Cristian Ramos-Vera, Angel García O'Diana, Miguel Basauri-Delgado, Yaquelin E Calizaya-Milla, and Jacksaint Saintila. 2024. Network analysis of anxiety and depressive symptoms during the COVID-19 pandemic in older adults in the United Kingdom. *Scientific Reports* 14, 1 (2024), 7741.

[51] Fergal Reid, Aaron McDaid, and Neil Hurley. 2012. Percolation computation in complex networks. In *2012 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*. IEEE, 274–281.

[52] Pedro Henrique Ribeiro Santiago, Gustavo Hermes Soares, Adrian Quintero, and Lisa Jamieson. 2022. Comparing the Clique Percolation with other overlapping community detection algorithms in psychological networks: A Monte Carlo simulation study. (2022).

[53] K Sathiyakumari and MS Vijaya. 2020. Overlapping Community Structure Detection using Twitter Data. *International Journal on Emerging Technologies* (2020).

[54] Karsten Steinhaeuser and Nitesh V Chawla. 2008. Community detection in a large real-world social network. In *Social computing, behavioral modeling, and prediction*. Springer, 168–175.

[55] 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.

[56] Ying-Wooi Wan, Rami Al-Ouran, Carl G Mangleburg, Thanneer M Perumal, Tom V Lee, Katherine Allison, Vivek Swarup, Cory C Funk, Chris Gaiteri, Mariet Allen, et al. 2020. Meta-analysis of the Alzheimer's disease human brain transcriptome and functional dissection in mouse models. *Cell reports* 32, 2 (2020).

[57] Jierui Xie, Stephen Kelley, and Boleslaw K Szymanski. 2013. Overlapping community detection in networks: The state-of-the-art and comparative study. *Acm computing surveys (csur)* 45, 4 (2013), 1–35.

[58] Xiaowei Ye, Rong-Hua Li, Qiangqiang Dai, Hongzhi Chen, and Guoren Wang. 2022. Lightning fast and space efficient k-clique counting. In *Proceedings of the ACM Web Conference 2022*. 1191–1202.

[59] Ziqi Ye, Yanmei Mu, Shianne Van Duzen, and Peter Ryser. 2024. Root and shoot phenology, architecture, and organ properties: an integrated trait network among 44 herbaceous wetland species. *New Phytologist* (2024).

[60] 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.

[61] Zhirong Yuan, You Peng, Peng Cheng, Li Han, Xuemin Lin, Lei Chen, and Wenjie Zhang. 2022. Efficient k-clique listing with set intersection speedup. *ICDE. IEEE* (2022).

[62] Xingyi Zhang, Congtao Wang, Yansen Su, Linqiang Pan, and Hai-Feng Zhang. 2017. A fast overlapping community detection algorithm based on weak cliques for large-scale networks. *IEEE Transactions on Computational Social Systems* 4, 4 (2017), 218–230.

[63] Di Zhuang, J Morris Chang, and Mingchen Li. 2019. DynaMo: Dynamic community detection by incrementally maximizing modularity. *IEEE Transactions on Knowledge and Data Engineering* 33, 5 (2019), 1934–1945.