The VLDB Journal (2016) 25:171–196
DOI 10.1007/s00778-015-0408-z

REGULAR PAPER

Diversified top-*k* clique search

Long Yuan[1] · Lu Qin[2] · Xuemin Lin[1] · Lijun Chang[1] · Wenjie Zhang[1]

Received: 9 April 2015 / Revised: 21 August 2015 / Accepted: 16 October 2015 / Published online: 30 October 2015

Abstract Maximal clique enumeration is a fundamental problem in graph theory and has been extensively studied. However, maximal clique enumeration is time-consuming in large graphs and always returns enormous cliques with large overlaps. Motivated by this, in this paper, we study the diversified top-*k* clique search problem which is to find top-*k* cliques that can cover most number of nodes in the graph. Diversified top-*k* clique search can be widely used in a lot of applications including community search, motif discovery, and anomaly detection in large graphs. A naive solution for diversified top-*k* clique search is to keep all maximal cliques in memory and then find *k* of them that cover most nodes in the graph by using the approximate greedy max *k*-cover algorithm. However, such a solution is impractical when the graph is large. In this paper, instead of keeping all maximal cliques in memory, we devise an algorithm to maintain *k* candidates in the process of maximal clique enumeration. Our algorithm has limited memory footprint and can achieve a guaranteed approximation ratio. We also introduce a novel light-weight PNP-Index, based on which we design an optimal maximal clique maintenance algorithm. We further explore three optimization strategies to avoid enumerating all maximal cliques and thus largely reduce the computational cost. Besides, for the massive input graph, we develop an I/O efficient algorithm to tackle the problem when the input graph cannot fit in main memory. We conduct extensive performance studies on real graphs and synthetic graphs. One of the real graphs contains 1.02 billion edges. The results demonstrate the high efficiency and effectiveness of our approach.

Keywords Graph · Diversified top-*k* search · Clique · I/O efficient

✉ Lu Qin
lu.qin@uts.edu.au

Long Yuan
longyuan@cse.unsw.edu.au

Xuemin Lin
lxue@cse.unsw.edu.au

Lijun Chang
ljchang@cse.unsw.edu.au

Wenjie Zhang
zhangw@cse.unsw.edu.au

1 The University of New South Wales, Sydney, Australia

2 Centre for QCIS, University of Technology, Sydney, Australia

1 Introduction

Maximal clique enumeration is a fundamental graph operation. Given an undirected graph G, a *clique* C is a subset of nodes in G in which every two nodes are connected by an edge, and C is a *maximal clique* if no superset of C is a clique. Maximal clique enumeration aims to enumerate all maximal cliques in G. Maximal clique enumeration has been extensively studied in the literature [3,11,13,15,16,22,23,37,40,43]. Maximal clique enumeration is known to be computationally intractable since the number of maximal cliques in a graph G can be exponential in the number of nodes in G [22]. It is difficult to process and analyze such a large number of maximal cliques. A possible solution is to compute top-*k* maximal cliques ranked by their size, since maximal cliques with larger size are more important and preferred for a user [8]. However, maximal cliques in a graph are usually highly overlapping [43], which significantly reduces the amount of useful information contained in the returned results. Motivated by this, in this paper, we study the problem of *diversified top-k clique search*, which aims to find

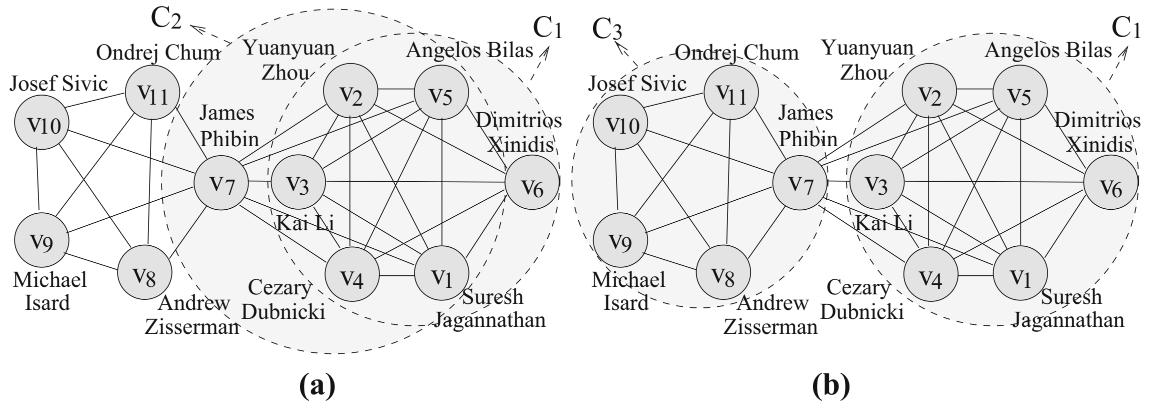

Fig. 1 Part of the collaboration network in *DBLP*

k cliques that are not only individually large but also lowly overlapping with each other.

Applications. Diversified top-k clique search can be applied in a wide range of applications. For example:

(1) Gene expression and motif discovery in molecular biology. In the gene co-expression network, Co-Expression Groups (CEGs) are represented as cliques [52]. Motif discovery in molecular biology requires to obtain the large CEGs with low overlaps, which can be modeled as the diversified top-k clique search problem.

(2) Anomaly detection in complex networks. In this application, cliques are used as signals of rare events and the problem is to find a set of large cliques with low overlaps [9], which can be modeled as the diversified top-k clique search problem.

(3) Community search in social networks. Diversified top-k cliques can serve as the seeds for community search, which can be expanded to lowly overlapping communities [30].

Specifically, given a graph G and an integer k, diversified top-k clique search is to find k cliques that are large and informative in the sense that they together cover most nodes in the graph. We illustrate this by the following example.

Example 1 Figure 1 shows part of the collaboration network in the *DBLP* dataset (http://www.informatik.uni-trier.de/~ley/db/), in which each node represents an author and each edge indicates the co-author relationship between two authors. There are three maximal cliques: $C_1 = \{v_1, v_2, v_3, v_4, v_5, v_6\}$, $C_2 = \{v_1, v_2, v_3, v_4, v_5, v_7\}$, and $C_3 = \{v_7, v_8, v_9, v_{10}, v_{11}\}$. Let $k = 2$. For top-k maximal cliques, we get the result $\mathcal{D}_1 = \{C_1, C_2\}$. For diversified top-$k$ cliques, the result is $\mathcal{D}_2 = \{C_1, C_3\}$. Although $|C_2| > |C_3|$, obviously, $\mathcal{D}_2$ is preferred to $\mathcal{D}_1$, since the two maximal cliques in $\mathcal{D}_1$ are highly overlapping with each other, while the two cliques in $\mathcal{D}_2$ cover all nodes in the graph.

Challenges. In order to compute diversified top-k cliques, a straightforward solution is to enumerate all maximal cliques first and then apply a greedy max k-cover algorithm [25] to compute an approximate result with a guaranteed approximation ratio. However, such a solution falls short to handle large graphs because clique enumeration is a costly operation and keeping all maximal cliques in memory is infeasible due to the exponential number of maximal cliques in a graph. Therefore, comparing to the conventional k-cover algorithm [25] and its variants (e.g. [31]), the following issues need to be addressed in order to make diversified top-k clique search practically applicable: (1) How to avoid generating all maximal cliques to compute the final result efficiently? (2) How to avoid keeping all generated maximal cliques in memory? and (3) How to guarantee the result quality when not all maximal cliques are generated and kept in memory? (4) How to solve the diversified top-k clique search problem when the input graph is too large to be loaded into main memory?

Contributions. In this paper, we answer all the above questions. The preliminary version is published in [48]. The main contributions of this work are summarized below.

(1) A simple problem model considering both size and diversity. We formalize the diversified top-k clique search as a problem to maximize the total number of nodes covered by the top-k cliques. The model is simple yet effective since it can be utilized to find large and diversified cliques simultaneously.

(2) An efficient algorithm with bounded memory consumption and a guaranteed approximation ratio. We devise an efficient algorithm based on the maximal clique enumeration algorithm to compute the diversified top-k cliques with a guaranteed approximation ratio. Our algorithm maintains at most k candidate maximal cliques in the memory and keeps updating the candidate set when more promising maximal cliques are generated. The key issue is how to efficiently update the candidate set when

new maximal cliques are generated. A basic solution requires $O(|\mathcal{A}| \cdot k \cdot |C_{max}|)$ time, which is costly, to maintain the candidate set, where $\mathcal{A}$ is the set of all maximal cliques, and C_{max} is the maximum clique in the graph. In this paper, we introduce a light-weight online PNP-Index. Based on the PNP-Index, we can reduce such time complexity to $O(\Sigma_{c \in \mathcal{A}}|C|)$, which is optimal in the sense that every generated maximal clique is accessed only once.

(3) Three novel pruning strategies with high pruning power. We explore three novel optimization strategies, namely, global pruning, local pruning, and initial candidate computation, to further improve the efficiency of our algorithm. Global pruning specifies a global search order of nodes during maximal clique computation, such that the algorithm can terminate as soon as expanding a certain node cannot improve the quality of current candidate set. Local pruning adopts a local pruning rule to avoid expanding unpromising partial cliques whenever possible. Initial candidate computation precomputes an initial candidate set using a greedy strategy before enumerating maximal cliques, such that both global pruning and local pruning methods can be applied more effectively. By applying the three optimization strategies, instead of generating all maximal cliques $\mathcal{A}$, our algorithm only needs to generate r maximal cliques, where $r << |\mathcal{A}|$, without sacrificing any result quality, thus the computational cost is largely reduced. Moreover, by using initial candidate computation, the result quality can be improved as well.

(4) An efficient algorithm with guaranteed I/O complexity. We develop an efficient algorithm with guaranteed I/O complexity to handle the scenario when the input graph cannot fit in main memory. We devise an oriented subgraph-based partition approach specialized for diversified top-k clique search problem and propose an I/O efficient algorithm which can utilize the limited main memory effectively and efficiently. Furthermore, our new proposed algorithm can achieve the same guaranteed approximation ratio as the in-memory algorithm.

(5) Extensive performance studies on real and synthetic datasets. We conduct extensive performance studies using real graphs and synthetic graphs . The experimental results demonstrate that our proposed algorithm can achieve both high effectiveness and high efficiency. Remarkably, for the in-memory approach, on a graph that contains 0.3 billion edges, our proposed algorithm can terminate in only one minute; for the I/O efficient approach, our approach can process massive graphs with billion-scale edges.

Outline. Section 2 provides the formal problem definition, shows the problem hardness, and presents two baseline solutions for the problem. Section 3 studies our new approach, introduces the novel PNP-Index, and proves its optimality. Section 4 explores three optimization strategies. Section 5 shows our I/O efficient algorithm for the large input graphs that cannot be held in main memory. Section 6 reviews the related work. Section 7 evaluates all introduced algorithms using extensive experiments, and Sect. 8 concludes the paper.

2 Preliminaries

2.1 Problem definition

We consider an undirected, unweighted, simple graph $G = (V, E)$, where $V(G)$ represents the set of nodes and $E(G)$ represents the set of edges in G. We denote the number of nodes and number of edges of graph G by n and m respectively, i.e., $n = |V(G)|$ and $m = |E(G)|$. For each node $u \in V(G)$, we use $id(u)$ to denote the id of node u , and use $N(u, G)$ to denote the set of neighbors of u in G, i.e., $N(u, G) = \{v|(u, v) \in E(G)\}$. The degree of a node $u \in V(G)$, denoted by $d(u, G)$, is the number of neighbors of u in G, i.e., $d(u, G) = |N(u, G)|$. For simplicity, we use $N(u)$ and $d(u)$ to denote $N(u, G)$ and $d(u, G)$, respectively, if the context is self-evident. A subgraph g of G is a graph such that $V(g) \subseteq V(G)$ and $E(g) \subseteq E(G)$. We use $g \subseteq G$ to denote that g is a subgraph of G.

Definition 1 (*Clique*) Given a graph G, a *clique* C in G is a set of nodes such that for any $u \in C$, $v \in C$ $(u \neq v)$, we have $(u, v) \in E(G)$. A clique C in G is called a *maximal clique* if there exists no clique C' in G such that $C \subset C'$.

Definition 2 (*Coverage* $\mathsf{cov}(\mathcal{D})$) Given a set of cliques $\mathcal{D} = \{C_1, C_2, \ldots\}$ in graph G, the *coverage* of $\mathcal{D}$, denoted by $\mathsf{cov}(\mathcal{D})$, is the set of nodes in G covered by the cliques in $\mathcal{D}$, i.e., $\mathsf{cov}(\mathcal{D}) = \bigcup_{C \in \mathcal{D}} C$.

Problem Statement. Given a graph G and an integer k, the problem of diversified top-k clique search is to compute a set $\mathcal{D}$, such that each $C \in \mathcal{D}$ is a clique, $|\mathcal{D}| \leq k$, and $|\mathsf{cov}(\mathcal{D})|$ is maximized. $\mathcal{D}$ is called diversified top-k cliques.

Problem Hardness. We show the hardness of the problem by considering the simple case: $k = 1$. In this case, the problem becomes the maximum clique problem which is NP-hard [26]. Therefore, *the diversified top-k clique search problem is an NP-hard problem*. In the literature, the fastest algorithm known to compute the maximum clique runs in time $O(2^{0.249n})$ [35].

2.2 Baseline solutions

In the literature, there are several algorithms to enumerate all maximal cliques in a graph G, and an algorithm to enumerate maximal cliques by considering the overlaps among

Algorithm 1 EnumAll(graph $G = (V, E)$, integer k)

```
1: $\mathcal{A} \leftarrow$ CliqueAll($V, \emptyset, \emptyset$);
2: return MaxCover($V, \mathcal{A}, k$);
3: procedure CliqueAll(node set $P$, node set $R$, node set $X$)
4: if $P \cup X = \emptyset$ then
5:     output $R$ as a maximal clique;
6: $u \leftarrow \text{argmax}_{v \in P \cup X}\{|P \cap N(v)|\}$;
7: for all $v \in P \setminus N(u)$ do
8:     CliqueAll($P \cap N(v), R \cup \{v\}, X \cap N(v)$);
9:     $P \leftarrow P \setminus \{v\}$; $X \leftarrow X \cup \{v\}$;
10: procedure MaxCover(set $V$, sets $\mathcal{S}$, integer $k$)
11: $\mathcal{D} \leftarrow \emptyset$; $V' \leftarrow V$;
12: for $i = 1$ to $k$ do
13:     $C \leftarrow \text{argmax}_{C' \in \mathcal{S} - \mathcal{D}}\{|C' \cap V'|\}$;
14:     $V' \leftarrow V' \setminus C$; $\mathcal{D} \leftarrow \mathcal{D} \cup \{C\}$;
15: return $\mathcal{D}$;
```

cliques. These algorithms lead to two baseline solutions for diversified top-k clique search. The first solution EnumAll enumerates all maximal cliques in the graph G, and then formulates the problem of diversified top-k clique search as a max k-cover problem which can be solved using a greedy strategy with a bounded approximation ratio. The second solution EnumSub computes a subset $\mathcal{S}$ of maximal cliques in G by considering the overlaps among cliques, and then applies the same greedy strategy as EnumAll to compute the diversified top-k cliques. However, the size of $\mathcal{S}$ cannot be bounded, and the result returned by EnumSub has no approximation guarantee.

2.3 Algorithm EnumAll

The EnumAll algorithm is shown in Algorithm 1. It first computes the set $\mathcal{A}$ of all maximal cliques using CliqueAll (line 1) and then computes the diversified top-k cliques using the greedy algorithm MaxCover (line 2).

Procedure CliqueAll. CliqueAll is the state-of-the-art algorithm for maximal clique enumeration introduced in [22]. It is a recursive backtracking algorithm based on three disjoint sets of nodes, P, R, and X (line 3). R is a partial maximal clique. P and X together cover the nodes that are connected to all nodes in R. The difference is that, P is the set of candidate nodes to be added into R to form larger cliques, and X contains the set of nodes that have been traversed to form maximal cliques in all previous levels of recursion. X is the set of nodes that must be excluded from R in order to avoid generating duplicated maximal cliques. Obviously, when $P \cup X$ is $\emptyset$, R is a maximal clique (line 4–5). In line 6, the algorithm finds a pivot node u that can maximize $|P \cap N(u)|$, and in line 7–9, the algorithm traverses all nodes v in $P \setminus N(u)$ (line 7), adds v into R recursively (line 8), and updates P and X (line 9). Note that the pivot node u computed in line 6 is used to reduce the number of candidate nodes to be added into R in each level of recursion by excluding $N(u)$ from P (line 7). As shown in [22], CliqueAll can be implemented in time $O(d \cdot n \cdot 3^{d/3})$ by specifying an order for nodes traversed in line 7, where $d \leq \sqrt{m}$ is the degeneracy of graph G with $d = \max_{g \subseteq G}\{\min_{v \in V(g)}\{d(v, g)\}\}$. It is proved in [22] that CliqueAll is worst-case optimal since there can be $\Theta((n - d) \cdot 3^{d/3})$ maximal cliques in the worst case.

Procedure MaxCover. The MaxCover algorithm (line 10-15) follows the greedy algorithm for the max k-cover problem, which is the problem of selecting k subsets from a collection of subsets such that their union contains as many elements as possible. Given the set of cliques $\mathcal{S}$, the nodes V of G, and an integer k, let $\mathcal{D}$ be the selected cliques, and V' be the set of nodes in V not covered by cliques in $\mathcal{D}$ (line 11). The algorithm greedily selects k cliques into $\mathcal{D}$ (line 12-14). Each time, the clique C to be selected is the one that can cover most nodes in V' (line 13). After selecting C, V' and $\mathcal{D}$ are updated accordingly (line 14). The MaxCover algorithm can achieve an approximation ratio of $(1-1/e) \approx 0.632$ which is the best-possible polynomial time approximation algorithm for the k-cover problem as shown in [25].

2.4 Algorithm EnumSub

The EnumSub algorithm is shown in Algorithm 2. Similar to EnumAll, EnumSub first computes a set $\mathcal{S}$ of maximal cliques in G using CliqueSub (line 1) and then invokes the greedy algorithm MaxCover to compute the diversified top-k cliques (line 2). The CliqueSub algorithm is proposed by Wang et al. [43] which computes a subset $\mathcal{S}$ of maximal cliques in a graph G by considering the overlaps among cliques. Suppose $\mathcal{A}$ is the set of all maximal cliques in G, it is guaranteed that for each maximal clique $C \in \mathcal{A}$, there exists a maximal clique $C' \in \mathcal{S}$, such that the similarity between C and C', denoted by $|C \cap C'|/|C|$, is no less than τ, for a parameter τ $(0 < \tau \leq 1)$.

The CliqueSub algorithm [43] follows the same framework of the CliqueAll algorithm (see the procedure CliqueAll in Algorithm 1). In [43], the authors observe that the CliqueAll algorithm typically outputs maximal cliques in an order such that two maximal cliques tend to be similar if they are near in the output sequence. Based on such an observation, the CliqueSub algorithm modifies the CliqueAll algorithm, such that each newly computed maximal clique C is reported only if its similarity to the previously reported maximal clique C' is small. In [43], a randomized algorithm and a deterministic algorithm are introduced to solve such a problem and some pruning rules are introduced to prune partial cliques at early stages of the CliqueSub algorithm.

Algorithm 2 EnumSub(graph $G = (V, E)$, integer k)

```
1: $\mathcal{S} \leftarrow$ CliqueSub($V, \emptyset, \emptyset$);
2: return MaxCover($V, \mathcal{S}, k$);
```

2.5 Limitations of baseline solutions

Comparing EnumAll (Algorithm 1) and EnumSub (Algorithm 2), EnumAll has a bounded approximation ratio; however, it needs to enumerate all maximal cliques; while EnumSub does not need to enumerate all maximal cliques, however, neither the number of reported maximal cliques nor the approximation ratio can be guaranteed. Both EnumAll and EnumSub are not scalable enough to handle large graphs due to the following two reasons:
(R_1) *Exponential number of cliques to be kept in memory.* Recall that the number of maximal cliques enumerated in EnumAll can achieve $\Theta((n-d)\cdot 3^{d/3})$ for a graph G with degeneracy d, which requires a huge amount of memory to keep all maximal cliques for the greedy algorithm MaxCover to compute the final top-k answers. Although the EnumSub algorithm can reduce the number of reported maximal cliques compared to EnumAll, it still outputs exponential number of maximal cliques without a bound. Therefore, as also verified by our experimental results in Sect. 7, EnumSub cannot essentially solve the problem when the graph is large.
(R_2) *Independent clique enumeration and top-k search processes.* The MaxCover algorithm, which is used in both EnumAll and EnumSub, adopts a global selection criteria in each iteration to greedily select the maximal clique that covers most new nodes in the graph as one of the top-k answers. Although MaxCover can achieve a bounded approximation ratio, the global selection criteria requires that all candidate maximal cliques should have been computed before invoking MaxCover. As a result, the maximal clique enumeration procedure (CliqueAll or CliqueSub) and the top-k search procedure (MaxCover) have to be invoked independently. However, if we maintain the top-k answers in the process of maximal clique enumeration, there will be more opportunities to prune the unpromising partial cliques at early stages of maximal clique enumeration and thus largely reduce the computational cost.

3 A new approach

In this paper, we devise a new algorithm for diversified top-k clique search, which can overcome the challenges introduced in Sect. 2.5. In the new algorithm, we modify the maximal clique enumeration algorithm CliqueAll (see the procedure CliqueAll in Algorithm 1) to integrate diversified top-k clique search into the process of maximal clique enumeration. Specifically, during the maximal clique enumeration process, we maintain k candidates of the most promising cliques that can maximize the total node coverage and update the k candidates when new maximal cliques are reported. The new algorithm has the following three advantages:

Algorithm 3 EnumKBasic(graph $G = (V, E)$, integer k)
```
1: $\mathcal{D} \leftarrow \emptyset$;
2: CliqueAll($V, \emptyset, \emptyset$) (replace line 5 in Algorithm 1 with
   CandMaintainBasic($R$));
3: return $\mathcal{D}$;
4: procedure CandMaintainBasic( clique $C$)
5: if $|\mathcal{D}| < k$ then { $\mathcal{D} \leftarrow \mathcal{D} \cup \{C\}$; return; }
6: $\mathcal{D}' \leftarrow (\mathcal{D} \setminus \{C_{min}(\mathcal{D})\}) \cup \{C\}$;
7: if $|\mathsf{priv}(C, \mathcal{D}')| > |\mathsf{priv}(C_{min}(\mathcal{D}), \mathcal{D})| + \alpha \times \frac{|\mathsf{cov}(\mathcal{D})|}{|\mathcal{D}|}$ then
8:     $\mathcal{D} \leftarrow \mathcal{D}'$;
```

(A_1) *Low memory consumption*. Unlike EnumAll and EnumSub, our algorithm does not need to keep all enumerated maximal cliques in memory. Instead, our algorithm just needs to keep the graph G and the k candidates of the most promising cliques in main memory, the size of which is much smaller than the size of all maximal cliques.
(A_2) *Guaranteed result quality*. Our algorithm can achieve a guaranteed approximation ratio of 0.25 and can be much better in practice as verified in the experiments (Sect. 7).
(A_3) *High pruning power*. By integrating diversified top-k clique search into the process of maximal clique enumeration, we can develop more pruning strategies to avoid expanding unpromising partial cliques at early stages of the maximal clique enumeration algorithm, thus largely improve the efficiency of the algorithm.

In this section, we introduce our basic algorithm and an improved algorithm which target at achieving A_1 and A_2. In the next section, we will focus on the optimization strategies such that A_3 can be achieved.

3.1 The basic algorithm

Before showing our basic algorithm EnumKBasic for diversified top-k clique search, we introduce some definitions.

Definition 3 (*Private-Node-Set* $\mathsf{priv}(C, \mathcal{D})$) Given a set of cliques $\mathcal{D} = \{C_1, C_2, \ldots\}$ in graph G, for each clique $C \in \mathcal{D}$, the *private-node-set* of C in $\mathcal{D}$, denoted by $\mathsf{priv}(C, \mathcal{D})$, is the set of nodes in C that are not contained in other cliques in $\mathcal{D}$, i.e., $\mathsf{priv}(C, \mathcal{D}) = C \setminus \mathsf{cov}(\mathcal{D} \setminus \{C\})$. Each $v \in \mathsf{priv}(C, \mathcal{D})$ is called a private node of C in $\mathcal{D}$.

Definition 4 (*Min-Cover-Clique* $C_{min}(\mathcal{D})$) Given a set of cliques $\mathcal{D} = \{C_1, C_2, \ldots\}$ in graph G, the *min-cover-clique* of $\mathcal{D}$, denoted by $C_{min}(\mathcal{D})$, is the clique $C \in \mathcal{D}$ with minimum $|\mathsf{priv}(C, \mathcal{D})|$, i.e., $C_{min}(\mathcal{D}) = \mathrm{argmin}_{C \in \mathcal{D}}\{|\mathsf{priv}(C, \mathcal{D})|\}$.

Algorithm EnumKBasic. Our basic algorithm Enum KBasic is shown in Algorithm 3. It first initializes the clique set $\mathcal{D}$, which is used to keep the candidates of diversified top-k cliques (line 1). Then it invokes the CliqueAll algorithm to enumerate maximal cliques without keeping all enumerated

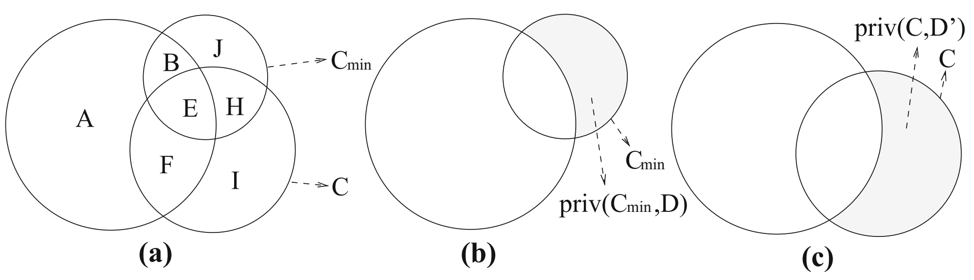

Fig. 2 Illustration for the Proof of Lemma 1

maximal cliques in memory. Instead, for each enumerated maximal clique C, the procedure CandMaintainBasic is invoked to update $\mathcal{D}$ using C (line 2). Finally, $\mathcal{D}$ is returned as the diversified top-k cliques (line 3).

The procedure CandMaintainBasic is shown in line 4–8 of Algorithm 3. When $|\mathcal{D}| < k$, $\mathcal{D}$ is updated by simply adding C (line 5). Otherwise, we try to replace $C_{min}(\mathcal{D})$ with C to generate a new clique set $\mathcal{D}'$ (line 6). If the number of private nodes of C in the new set $\mathcal{D}'$ is larger than the number of private nodes of $C_{min}(\mathcal{D})$ in $\mathcal{D}$ by $\alpha \times \frac{|\mathsf{cov}(\mathcal{D})|}{|\mathcal{D}|}$ for a parameter α ($0 < \alpha \leq 1$), then $\mathcal{D}$ is updated to be $\mathcal{D}'$ (line 7-8).

Algorithm Analysis. Obviously, Algorithm 3 only needs to keep graph G and the candidate set $\mathcal{D}$ in main memory. Next, we analyze the time cost of Algorithm 3. Suppose C_{max} is the maximum clique in G, and $\mathcal{A}$ is the set of all maximal cliques in G, in procedure CandMaintainBasic, we need to compute $C_{min}(\mathcal{D})$, $|\mathsf{priv}(C, \mathcal{D}')|$, $|\mathsf{priv}(C_{min}(\mathcal{D}), \mathcal{D})|$, and $|\mathsf{cov}(\mathcal{D})|$. It is easy to prove that each of the four values can be computed in time $O(k \cdot |C_{max}|)$ by traversing nodes in C and all cliques in $\mathcal{D}$ only once. Suppose $T_{enum}(G)$ is the time to enumerate all maximal cliques in G, the time complexity of Algorithm 3 is $O(T_{enum}(G) + |\mathcal{A}| \cdot k \cdot |C_{max}|)$.

The following lemma shows the quality of the result for the diversified top-k cliques computed using Algorithm 3.

Lemma 1 *Given a graph G and an integer k, suppose $\mathcal{D}^*$ is the optimal diversified top-k cliques, and $\mathcal{D}$ is the diversified top-k cliques returned by Algorithm 3 with $\alpha = 1$, we have* $|\mathsf{cov}(\mathcal{D})| \geq 0.25 \times |\mathsf{cov}(\mathcal{D}^*)|$.

Proof Sketch The proof is based on a theoretical result derived in [5], which shows the following result:

Given a stream of sets $\mathcal{A} = \{C_1, C_2, \ldots\}$, and an integer k, let $\mathcal{A}_i = \{C_1, C_2, \ldots, C_i\}$ for $i > 0$, and $\mathcal{D}_i = \mathcal{A}_i$ for $0 < i \leq k$. For any $i > k$, we construct $\mathcal{D}_i$ from $\mathcal{D}_{i-1}$ as follows: suppose $\mathcal{D}_i' = (\mathcal{D}_{i-1} \setminus \{C_{min}(\mathcal{D}_{i-1})\}) \cup \{C_i\}$, then $\mathcal{D}_i = \mathcal{D}_i'$ if $|\mathsf{cov}(\mathcal{D}_i')| > (1 + \frac{1}{k})|\mathsf{cov}(\mathcal{D}_{i-1})|$, and $\mathcal{D}_i = \mathcal{D}_{i-1}$ otherwise. It can be guaranteed that $\mathcal{D}_i$ is a 0.25-approximation solution of the max k-cover problem on $\mathcal{A}_i$.

The above problem has the same setting as our problem. Thus, we only need to prove that when $\alpha = 1$, the condition in line 7 of Algorithm 3, $|\mathsf{priv}(C, \mathcal{D}')| > |\mathsf{priv}(C_{min}(\mathcal{D}), \mathcal{D})| + \frac{|\mathsf{cov}(\mathcal{D})|}{|\mathcal{D}|}$, is equivalent to the condition $|\mathsf{cov}(\mathcal{D}')| > (1 + \frac{1}{k})|\mathsf{cov}(\mathcal{D})|$ used in [5], for $\mathcal{D}' = (\mathcal{D} \setminus \{C_{min}(\mathcal{D})\}) \cup \{C\}$. Let $C_{min} = C_{min}(\mathcal{D})$, the relationship of $\mathcal{D}$, $\mathcal{D}'$, C, and C_{min} is illustrated in Fig. 2 using the seven subsets A, B, E, F, J, H, and I. Obviously, the condition $|\mathsf{cov}(\mathcal{D}')| > (1+\frac{1}{k})|\mathsf{cov}(\mathcal{D})|$ is equivalent to $|A| + |B| + |E| + |F| + |H| + |I| > (1 + \frac{1}{k})(|A| + |B| + |E| + |F| + |H| + |J|)$, which is equivalent to $|H| + |I| > |H| + |J| + \frac{1}{k} \times |\mathsf{cov}(\mathcal{D})|$. Since $|\mathcal{D}| = k$, we have $|\mathsf{priv}(C, \mathcal{D}')| > |\mathsf{priv}(C_{min}, \mathcal{D})| + \frac{|\mathsf{cov}(\mathcal{D})|}{|\mathcal{D}|}$. □

Discussion. Note that when $\alpha = 1$, Algorithm 3 is essentially the same as the approach introduced in [5]. However, in this paper, we use a parameter α to allow more flexibility in choosing the size of the newly added cliques to replace the min-cover-clique. Generally, a smaller α will lead to more covered nodes and result in a higher computational cost in the meantime. However, there are no theoretical results on how to set the best α. From the results of our experiments, 0.3 is a good candidate considering the tradeoff between the efficiency and effectiveness of the algorithm. In addition, in [5], the algorithm needs to compute $|\mathsf{cov}(\mathcal{D}')|$, which is costly, while in Algorithm 3, we only use $|\mathsf{priv}(C, \mathcal{D}')|$ and $|\mathsf{priv}(C_{min}(\mathcal{D}), \mathcal{D})|$, which can lead to an efficient algorithm with the help of an online Private-Node-set Preserved Index (PNP-Index), which will be introduced in the next subsection.

3.2 Optimal candidate maintenance

As discussed in Sect. 3.1, in addition to the time $O(T_{enum}(G))$ to enumerate all the maximal cliques $\mathcal{A}$, Algorithm 3 needs extra $O(|\mathcal{A}| \cdot k \cdot |C_{max}|)$ time to maintain the candidate set $\mathcal{D}$. It is highly possible that $O(|\mathcal{A}| \cdot k \cdot |C_{max}|) > O(T_{enum}(G))$ when either k or $|C_{max}|$ is large. Therefore, the algorithm is inefficient. The main cost lies on computing the values of $C_{min}(\mathcal{D})$, $|\mathsf{priv}(C, \mathcal{D}')|$, $|\mathsf{priv}(C_{min}(\mathcal{D}), \mathcal{D})|$, and $|\mathsf{cov}(\mathcal{D})|$, each of which needs to traverse all nodes of the cliques in $\mathcal{D}$ in the worst case. In this subsection, we introduce a novel online *Private-Node-set Preserved Index* (PNP-Index), which is used to maintain $|\mathsf{priv}(C, \mathcal{D})|$ for each $C \in \mathcal{D}$, such that $C_{min}(\mathcal{D})$, $|\mathsf{priv}(C, \mathcal{D}')|$, $|\mathsf{priv}(C_{min}(\mathcal{D}), \mathcal{D})|$, and $|\mathsf{cov}(\mathcal{D})|$ can be computed efficiently. We will show that the PNP-Index is compact which only consumes $O(\Sigma_{C \in \mathcal{D}}|C|)$ space, and with PNP-Index, EnumK only takes $O(T_{enum}(G))$

time, which is optimal in the sense that no extra cost is introduced in the time complexity when maintaining $\mathcal{D}$.

Definition 5 (*Reverse Coverage* $\mathsf{rcov}(v, \mathcal{D})$) Given a set of cliques $\mathcal{D}$ in graph G, for each node $v \in V(G)$, the *reverse coverage* of v, denoted by $\mathsf{rcov}(v, \mathcal{D})$, is the set of cliques in $\mathcal{D}$ that contain v, i.e., $\mathsf{rcov}(v, \mathcal{D}) = \{C | v \in C, C \in \mathcal{D}\}$.

Definition 6 (*Reverse Private-Node-Set* $\mathsf{rpriv}(i, \mathcal{D})$) Given a set of cliques $\mathcal{D}$ in graph G, for each integer $0 \le i \le |V(G)|$, the *reverse private-node-set* of i, denoted by $\mathsf{rpriv}(i, \mathcal{D})$, is the set of cliques C in $\mathcal{D}$ such that $|\mathsf{priv}(C, \mathcal{D})| = i$, i.e., $\mathsf{rpriv}(i, \mathcal{D}) = \{C | C \in \mathcal{D}, |\mathsf{priv}(C, \mathcal{D})| = i\}$.

The PNP-Index. In the following, for simplicity, we use cov, $\mathsf{priv}(C)$, C_{min}, $\mathsf{rcov}(v)$, and $\mathsf{rpriv}(i)$ to denote $\mathsf{cov}(\mathcal{D})$, $\mathsf{priv}(C, \mathcal{D})$, $C_{min}(\mathcal{D})$, $\mathsf{rcov}(v, \mathcal{D})$, and $\mathsf{rpriv}(i, \mathcal{D})$ respectively, if the context is self-evident. The PNP-Index is built on $\mathcal{D}$ and graph G. In addition to $\mathcal{D}$, the PNP-Index consists of five additional components:

- $|\mathsf{priv}(C)|$: the number of private nodes for each $C \in \mathcal{D}$.
- $\mathsf{rcov}(v)$: the reverse coverage for each $v \in V(G)$.
- $|\mathsf{cov}|$: the number of nodes covered by $\mathcal{D}$.
- C_{min}: the clique C in $\mathcal{D}$ with minimum $|\mathsf{priv}(C)|$.
- $\mathsf{rpriv}(i)$: the reverse private-node-set for $0 \le i \le |V(G)|$.

Where $\mathsf{rcov}(v)$ is used to check whether the node v is privately contained in a certain clique in $\mathcal{D}$, and $\mathsf{rpriv}(i)$ is used to update C_{min} in time $O(|C|)$ (which is independent to k) when processing a newly generated clique C.

Algorithm EnumK. The new algorithm EnumK is shown in Algorithm 4, which follows the same framework of Algorithm 3 with different candidate maintenance procedures. The new procedure CandMaintain is shown in line 5–9. For each generated clique C, when $|\mathcal{D}| < k$, it inserts C into $\mathcal{D}$ by invoking a procedure Insert(C) (line 6). Otherwise, it calculates p_{new} which is $|\mathsf{priv}(C, \mathcal{D}')|$ for $\mathcal{D}' = (\mathcal{D} \backslash \{C_{min}\}) \cup \{C\}$ (line 7). If the update condition is satisfied, $\mathcal{D}$ is updated by deleting C_{min} using Delete(C_{min}) and inserting C using Insert(C) (line 8–9). The key procedures are how to calculate $p_{new} = |\mathsf{priv}(C, \mathcal{D}')|$ (line 7) and how to maintain the PNP-Index using Delete and Insert. Below, we show an example to illustrate the EnumK algorithm and the PNP-Index, and then we introduce the details of the three key procedures.

Example 2 Figure 3a shows a graph G with 12 nodes. Suppose $k = 3$ and the current candidate set is $\mathcal{D} = \{C_1, C_2, C_3\}$, we have $C_{min} = C_2$ with $|\mathsf{priv}(C_{min})| = |\{v_6\}| = 1$, and $|\mathsf{cov}| = 10$. Let $C_4 = \{v_6, v_7, v_9, v_{11}, v_{12}\}$ be the next clique generated, then $\mathcal{D}' = (\mathcal{D} \setminus \{C_2\}) \cup C_4 = \{C_1, C_3, C_4\}$. We have $p_{new} = |\mathsf{priv}(C_4, \mathcal{D}')| = |\{v_6, v_{11}, v_{12}\}| = 3$. Suppose

Algorithm 4 EnumK(graph $G = (V, E)$, integer k)
```
1: $\mathcal{D} \leftarrow \emptyset$; $|\mathsf{cov}| \leftarrow 0$; $C_{min} \leftarrow \emptyset$;
2: $\mathsf{rcov}(v) \leftarrow \emptyset$ for all $v \in V(G)$; $\mathsf{rpriv}(i) \leftarrow \emptyset$ for all $0 \le i \le |V(G)|$;
3: CliqueAll$(V, \emptyset, \emptyset)$ (replace line 5 in Algorithm 1 with CandMaintain$(R)$);
4: return $\mathcal{D}$;
5: procedure CandMaintain( clique $C$)
6: if $|\mathcal{D}| < k$ then { Insert$(C)$; return; }
7: $p_{new} \leftarrow |\{v \in C$ s.t. $|\mathsf{rcov}(v)| = 0$ or ($|\mathsf{rcov}(v)| = 1$ and $v \in C_{min}$) $\}|$;
8: if $p_{new} > |\mathsf{priv}(C_{min})| + \alpha \times \frac{|\mathsf{cov}|}{|\mathcal{D}|}$ then
9:   Delete$(C_{min})$; Insert$(C)$;
10: procedure Delete( clique $C$)
11: $\mathcal{D} \leftarrow \mathcal{D} \setminus \{C\}$; remove $C$ from $\mathsf{rpriv}(|\mathsf{priv}(C)|)$;
12: for all $v \in C$ do
13:   $\mathsf{rcov}(v) \leftarrow \mathsf{rcov}(v) \setminus \{C\}$;
14:   if $|\mathsf{rcov}(v)| = 0$ then $|\mathsf{cov}| \leftarrow |\mathsf{cov}| - 1$;
15:   if $|\mathsf{rcov}(v)| = 1$ then
16:     $C' \leftarrow$ the maximal clique in $\mathsf{rcov}(v)$;
17:     $|\mathsf{priv}(C')| \leftarrow |\mathsf{priv}(C')| + 1$;
18:     move $C'$ from $\mathsf{rpriv}(|\mathsf{priv}(C')| - 1)$ to $\mathsf{rpriv}(|\mathsf{priv}(C')|)$;
19: procedure Insert( clique $C$)
20: $\mathcal{D} \leftarrow \mathcal{D} \cup \{C\}$; $|\mathsf{priv}(C)| \leftarrow 0$;
21: for all $v \in C$ do
22:   $\mathsf{rcov}(v) \leftarrow \mathsf{rcov}(v) \cup \{C\}$;
23:   if $|\mathsf{rcov}(v)| = 1$ then
24:     $|\mathsf{priv}(C)| \leftarrow |\mathsf{priv}(C)| + 1$; $|\mathsf{cov}| \leftarrow |\mathsf{cov}| + 1$;
25:   if $|\mathsf{rcov}(v)| = 2$ then
26:     $C' \leftarrow$ the clique in $\mathsf{rcov}(v) \setminus \{C\}$;
27:     $|\mathsf{priv}(C')| \leftarrow |\mathsf{priv}(C')| - 1$;
28:     move $C'$ from $\mathsf{rpriv}(|\mathsf{priv}(C')| + 1)$ to $\mathsf{rpriv}(|\mathsf{priv}(C')|)$;
29: $\mathsf{rpriv}(|\mathsf{priv}(C)|) \leftarrow \mathsf{rpriv}(|\mathsf{priv}(C)|) \cup \{C\}$;
30: for $i = 0$ to $|\mathsf{priv}(C)|$ do
31:   if $\mathsf{rpriv}(i) \ne \emptyset$ then
32:     $C_{min} \leftarrow$ an arbitrary clique in $\mathsf{rpriv}(i)$;
33:     break;
```

$\alpha = 0.5$, we have $p_{new} = 3 > |\mathsf{priv}(C_{min})| + \alpha \times \frac{|\mathsf{cov}|}{|\mathcal{D}|} = 2.67$. Therefore, after processing C_4, $\mathcal{D}$ is updated to be $\{C_1, C_3, C_4\}$ as shown in Fig. 3b.

Figure 4 shows the corresponding PNP-Index before and after processing C_4, in which each row corresponds to a clique C and each column corresponds to a node v in G. We also show $|\mathsf{priv}(C)|$ for each clique C in the last column, $|\mathsf{rcov}(v)|$ for each node v in the last row, and $|\mathsf{cov}|$ in the bottom right cell. The columns for private nodes are colored gray. For each $v \in C$, the corresponding cell is marked $\hat{1}$ if $v \in \mathsf{priv}(C)$, and 1 otherwise. C_{min} is marked a star (*) in the corresponding cell of the last column.

Computing $\mathsf{priv}(C, \mathcal{D}')$. $\mathsf{priv}(C, \mathcal{D}')$ consists of two parts:

(P_1) The nodes that are contained in C but not contained in $\mathsf{cov}(\mathcal{D})$, e.g., the part denoted by I in Fig. 2. This part can be calculated using $|\{v \in C$ s.t. $|\mathsf{rcov}(v)| = 0\}|$.

(P_2) The nodes that are contained in both C and $\mathsf{priv}(C_{min}, \mathcal{D})$, e.g., the part denoted by H in Fig. 2. This

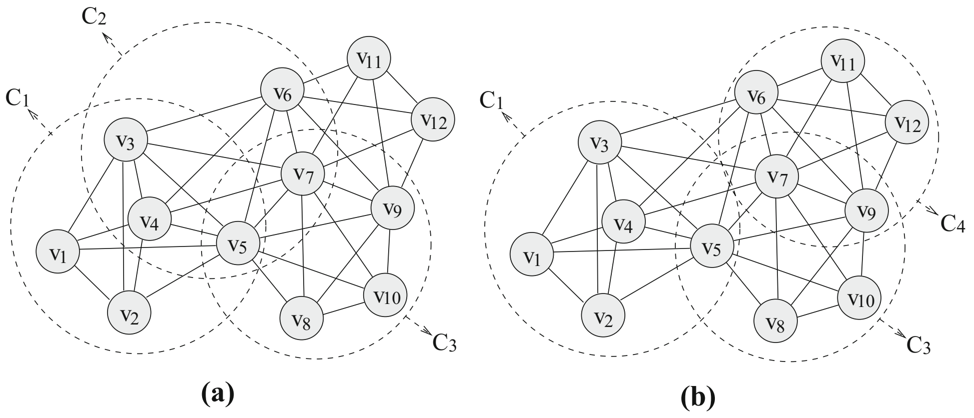

Fig. 3 Candidate maintenance on graph G

Fig. 4 Example of the PNP-Index

$C \setminus v$	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}	$\lvert$priv$(C)\rvert$
Before processing C_4:													
C_1	$\hat{1}$	$\hat{1}$	1	1	1								2
C_2			1	1	1	$\hat{1}$	1						1^*
C_3					1		1	$\hat{1}$	$\hat{1}$	$\hat{1}$			3
$\lvert$rcov$(v)\rvert$	1	1	2	2	3	1	2	1	1	1			$\lvert$cov$\rvert$:10
C_4						1	1		1		1	1	
After processing C_4:													
C_1	$\hat{1}$	$\hat{1}$	$\hat{1}$	$\hat{1}$	1								4
C_3					1		1	$\hat{1}$	1	$\hat{1}$			2^*
C_4						$\hat{1}$	1		1		$\hat{1}$	$\hat{1}$	3
$\lvert$rcov$(v)\rvert$	1	1	1	1	2	1	2	1	2	1	1	1	$\lvert$cov$\rvert$:12

part can be calculated using $|\{v \in C$ s.t. $|\mathsf{rcov}(v)| = 1$ and $v \in C_{min})\}|$.

In summary, $\mathsf{priv}(C, \mathcal{D}')$ (or p_{new}) can be calculated as $|\{v \in C$ s.t. $|\mathsf{rcov}(v)| = 0$ or $(|\mathsf{rcov}(v)| = 1$ and $v \in C_{min})$ $\}|$, as shown in line 7 of Algorithm 4.

Example 3 We show how to compute $p_{new} = \mathsf{priv}(C, \mathcal{D}')$ in Example 2 when processing $C = C_4$. The part P_1 can be computed as $|\{v_{11}, v_{12}\}| = 2$, and the part P_2 can be computed as $|\{v_6\}| = 1$, since $v_6 \in C_{min} = C_2$ and $|\mathsf{rcov}(v_6)| = 1$, as shown in Fig. 4. As a result, $\mathsf{priv}(C, \mathcal{D}') = 2 + 1 = 3$.

Procedure Delete(C). After removing an existing clique C from $\mathcal{D}$, we also need to maintain $\mathsf{rcov}(v)$, $|\mathsf{cov}|$, and $\mathsf{rpriv}(i)$ (for some nodes v and integers i) whose values are changed due to the deletion of C. This is processed by the procedure Delete(C), which is shown in line 10–18 of Algorithm 4.

After removing C from both $\mathcal{D}$ and $\mathsf{rpriv}(|\mathsf{priv}(C)|)$ (line 11), the algorithm visits all nodes $v \in C$ (line 12), and for each such v, it processes the updates in line 13–18. Since C covers node v, $\mathsf{rcov}(v)$ is updated by removing C (line 13). After updating $\mathsf{rcov}(v)$, there are two cases to be considered:

(Case 1) $|\mathsf{rcov}(v)|$ *decreases from* 1 *to* 0: This case indicates that a existing node v which is privately covered by C is not covered by any cliques after removing C, thus, $|\mathsf{cov}|$ decreases by 1 (line 14).
(Case 2) $|\mathsf{rcov}(v)|$ *decreases from* 2 *to* 1: In this case, v becomes a private node for the clique C', where C' is the only clique that contains v after removing C (line 16). Thus $|\mathsf{priv}(C')|$ is updated by increasing 1 (line 17), and C' is removed from $\mathsf{rpriv}(|\mathsf{priv}(C')| - 1)$and added into $\mathsf{rpriv}(|\mathsf{priv}(C')|)$ (line 18).
In other cases when $|\mathsf{rcov}(v)|$ decreases to be larger than 1, no other updates will be triggered.

Example 4 Continue Example 3. Before adding C_4 into $\mathcal{D}$, we need to remove $C_{min} = C_2$ from $\mathcal{D}$. After removing C_2, v_6 is not covered by any other cliques (the case for $|\mathsf{rcov}(v_6)| = 0$), and thus $|\mathsf{cov}|$ decreases by 1 (line 14); v_3 is only covered by $C_1 \in \mathsf{rcov}(v_3)$ (the case for $|\mathsf{rcov}(v_3)| = 1$), and thus $|\mathsf{priv}(C_1)|$ increases by 1 (line 17).

Procedure $\mathsf{Insert}(C)$. After inserting a new clique C into $\mathcal{D}$, we also need to maintain $\mathsf{priv}(C')$, $\mathsf{rcov}(v)$, $|\mathsf{cov}|$, $\mathsf{rpriv}(i)$, and C_{min} (for some cliques C', nodes v, and integers i) whose values are changed due to the insertion of C. This is processed by the procedure $\mathsf{Insert}(C)$, which is shown in line 19–33 of Algorithm 4.

After inserting C into $\mathcal{D}$ and initializing $|\mathsf{priv}(C)|$ to be 0 (line 20), the algorithm visits all nodes $v \in C$ (line 21), and for each such v, it processes the updates in line 22–28. Since C covers node v, $\mathsf{rcov}(v)$ is updated by adding C (line 20). After updating $\mathsf{rcov}(v)$, there are two cases to be considered:

(Case 1) $|\mathsf{rcov}(v)|$ *increases from* 0 *to* 1: This case indicates that a new node v is privately covered by C, thus, both $|\mathsf{priv}(C)|$ and $|\mathsf{cov}|$ increase by 1 (line 23–24).
(Case 2) $|\mathsf{rcov}(v)|$ *increases from* 1 *to* 2: In this case, v is removed from the private node set of a clique C', where C' is the only clique that contains v before adding C (line 26). Thus $|\mathsf{priv}(C')|$ is updated by decreasing 1 (line 27), and C' is removed from $\mathsf{rpriv}(|\mathsf{priv}(C')| + 1)$ and added into $\mathsf{rpriv}(|\mathsf{priv}(C')|)$ (line 28).

In other cases when $|\mathsf{rcov}(v)|$ increases to be larger than 2, no other updates will be triggered. After traversing all nodes in C, $|\mathsf{priv}(C)|$ is computed. Thus, we need to update $\mathsf{rpriv}(|\mathsf{priv}(C)|)$ by adding a new element C (line 29). Finally, we need to update C_{min} as follows.

Updating C_{min}: A straightforward solution to update C_{min} is to search the clique C' with minimum $\mathsf{priv}(C')$ from $\mathcal{D}$. However, such an operation may introduce an extra cost which is dependent on the value of k, making the algorithm inefficient when k is large. Note that we have maintained the sets $\mathsf{rpriv}(i)$ for all possible i. With all $\mathsf{rpriv}(i)$ $(1 \le i \le |V|)$, $|\mathsf{priv}(C_{min})|$ is the first element j such that $\mathsf{rpriv}(j) \neq \emptyset$. We also have $|\mathsf{priv}(C_{min})| \le |\mathsf{priv}(C)|$. Therefore, we can try every i from 0 to $|\mathsf{priv}(C)|$ (line 30) and find C_{min} in $\mathsf{rpriv}(i)$ for the first i with $\mathsf{rpriv}(i) \neq \emptyset$ (line 31). In this way, C_{min} can be computed in $O(|\mathsf{priv}(C_{min})|) \le O(|\mathsf{priv}(C)|) \le O(|C|)$ time which is independent to k.

Example 5 Continue Example 4. After deleting $C_{min} = C_2$ from $\mathcal{D}$, we insert C_4 into $\mathcal{D}$. After inserting C_4, v_{11} is only covered by C_4 (the case for $|\mathsf{rcov}(v_{11}| = 1)$, and thus both $|\mathsf{priv}(C_4)|$ and $|\mathsf{cov}|$ increase by 1 (line 24); v_9 is covered by two cliques $C_3 \in \mathsf{rcov}(v_9)$ and $C_4 \in \mathsf{rcov}(v_9)$ (the case for $|\mathsf{rcov}(v_9| = 2)$, and thus $|\mathsf{priv}(C_3)|$ decreases by 1 (line 27) indicating that v_9 is not a private node for C_3.

Optimality. The following two lemmas show the optimality of Algorithm 4. Lemma 2 indicates that the space complexity of the $\mathsf{PNP\text{-}Index}$ is the same as $\mathcal{D}$. Lemma 3 shows that the time complexity of Algorithm 4 is the same as that of maximal clique enumeration ($\mathsf{CliqueAll}$ in Algorithm 1). In other words, the maintenance of $\mathcal{D}$ using $\mathsf{PNP\text{-}Index}$ does not take extra cost w.r.t. both space and time complexities.

Lemma 2 *The* $\mathsf{PNP\text{-}Index}$ *uses* $O(\Sigma_{C\in\mathcal{D}}|C|)$ *memory.*

Proof Sketch Each $C \in \mathcal{D}$ can be stored as a hash set with $O(|C|)$ space s.t. for any $v \in V(G)$, $v \in C$ can be detected in $O(1)$ time. The reverse coverage $\mathsf{rcov}(v)$ for all $v \in V(G)$ consumes $O(\Sigma_{C\in\mathcal{D}}|C|))$ space. The reverse private-node-set $\mathsf{rpriv}(i)$ for all $0 \le i \le |V(G)|$ consumes $O(|\mathcal{D}|)$ space by only keeping those $\mathsf{rpriv}(i) \neq \emptyset$ in memory, and $|\mathsf{priv}(C)|$ for all $C \in \mathcal{D}$ consumes $O(|\mathcal{D}|)$ space. In summary, the total memory used by $\mathsf{PNP\text{-}Index}$ is $O(\Sigma_{C\in\mathcal{D}}|C|)$. □

Lemma 3 *The time complexity of Algorithm 4 is* $O(T_{enum}(G))$*, where* $O(T_{enum}(G))$ *is the time to enumerate all maximal cliques in* G*.*

Proof Sketch The main cost of Algorithm 4 is spent on the three key procedures used in $\mathsf{CandMaintain}$ to process maximal clique C, namely, computing $p_{new} = \mathsf{priv}(C, \mathcal{D}')$, $\mathsf{Delete}(C)$, and $\mathsf{Insert}(C)$. Computing p_{new} takes $O(|C|)$ time, and $\mathsf{Delete}(C)$ takes $O(|C|)$ time by traversing every $v \in C$ only once. $\mathsf{Insert}(C)$ also takes $O(|C|)$ time by traversing every $v \in C$ only once in line 21–28, and then computing C_{min} in $O(|C|)$ time in line 30–33. As a result, processing each maximal clique C takes time $O(|C|)$ which is the same as the time of outputting C. Therefore, the time complexity of Algorithm 4 is dominated by $O(T_{enum}(G))$. □

4 Optimization strategies

4.1 Solution overview

Recall that Algorithm 4 computes diversified top-k cliques by processing each generated clique only once. For each new clique C, it tries to use C to replace the clique with the least number of private nodes in the current candidate set $\mathcal{D}$. Algorithm 4 is efficient to maintain $\mathcal{D}$ using $\mathsf{PNP\text{-}Index}$. However, it does not consider the possible opportunities to reduce the set of cliques to be enumerated, and thus reduce the total computational cost. Therefore, in this section, we find three strategies, namely, global pruning, local pruning, and initial candidate computation, to further optimize Algorithm 4 with the aim of reducing the total number of cliques enumerated by Algorithm 4 without reducing the quality of the final answers. Our optimization strategies are based on the following three observations.

(Observation 1) Global pruning: We assign a global priority for all nodes in the graph G when enumerating maximal cliques in Algorithm 4, such that the nodes

Algorithm 5 EnumKOpt(graph $G = (V, E)$, integer k)

```
1: line 1-3 of Algorithm 4;
2: $\mathcal{D} \leftarrow$ InitK$(G, k)$;
3: $P \leftarrow V$; $R \leftarrow \emptyset$; $X \leftarrow \emptyset$; $u \leftarrow \text{argmax}_{v \in V}\{d(v)\}$;
4: for all $v \in V \setminus N(u)$ in non-increasing order of score$(v)$ do
5:     if $|\mathcal{D}| = k$ and GlobalPruning$(v)$ then break;
6:     CliqueK$(P \cap N(v), R \cup \{v\}, X \cap N(v))$;
7:     $P \leftarrow P \setminus \{v\}$; $X \leftarrow X \cup \{v\}$;
8: return $\mathcal{D}$;
9: procedure CliqueK$(P, R, X)$
10: if $P \cup X = \emptyset$ then CandMaintain$(R)$;
11: if LocalPruning$(P, R)$ then return;
12: $u \leftarrow \text{argmax}_{v \in P \cup X}\{|P \cap N(v)|\}$;
13: for all $v \in P \setminus N(u)$ do
14:     CliqueK$(P \cap N(v), R \cup \{v\}, X \cap N(v))$;
15:     $P \leftarrow P \setminus \{v\}$; $X \leftarrow X \cup \{v\}$;
```

with high potential to result in large maximal cliques are expanded first. Then the algorithm can terminate early when expanding the remaining nodes does not improve the quality of the current candidates.

(Observation 2) Local pruning: For a partial clique R computed in Algorithm 4, if R has no potential to be expanded to improve the quality of the current candidates, the whole branch expanded from R in Algorithm 4 can be pruned.

(Observation 3) Initial candidate computation: We compute a good initial candidate set $\mathcal{D}$ of k cliques before enumerating all cliques in Algorithm 4. Then both global pruning and local pruning conditions can be satisfied earlier.

Our optimized algorithm EnumKOpt is shown in Algorithm 5, which follows the same framework of Algorithm 4 by adding the three optimization strategies. After initialization (line 1), the algorithm computes the initial candidate set $\mathcal{D}$ using InitK(G, k) (Sect. 4.4). Then the algorithm traverses the nodes $v \in V$ in non-increasing order of their priorities, denoted by score(v) (line 4), and stops when the global pruning condition (Sect. 4.2) is satisfied (line 5). Otherwise, the algorithm invokes CliqueK to enumerate maximal cliques expanded from v (line 6). CliqueK (line 9–15) follows the same framework of CliqueAll in Algorithm 1 by adding the local pruning rule (line 11) and replacing line 5 of Algorithm 1 with CandMaintain(R) which uses the PNP-Index to efficiently maintain the candidate set $\mathcal{D}$ (refer to Algorithm 4). In the following, we will introduce the three optimization strategies.

4.2 Global pruning

In global pruning, we need to calculate the priority score(v) for each node $v \in V(G)$ efficiently, and derive a global pruning condition that makes use of score(v) such that the algorithm can terminate as early as possible. Recall that score(v) represents the potential size of the maximum clique expanded from v. Thus, the best way is to use the clique number $\omega(v)$ as score(v) based on the following definition:

Definition 7 (*Clique Number $\omega(v)$ and $\omega(S)$*) Given a graph G and a node $v \in V(G)$, the clique number of v in G, denoted by $\omega(v)$, is the size of the maximum clique C in G that contains v, i.e., $\omega(v) = \max_{C \in \mathcal{A}(G), v \in C}\{|C|\}$, where $\mathcal{A}(G)$ is the set of maximal cliques in G. Given a set of nodes $S \subseteq V(G)$, the clique number of S, denoted by $\omega(S)$, is the size of the maximum clique C such that $C \subseteq S$, i.e., $\omega(S) = \max_{C \subseteq S, C \text{ is a clique}}\{|C|\}$. Obviously, $\omega(v) = \omega(N(v)) + 1$.

However, as shown in Sect. 2.1, finding the maximum clique in G is an NP-hard problem, and thus computing $\omega(v)$ for all $v \in V(G)$ is also an NP-hard problem. Thus, instead of using $\omega(v)$, we use an upper bound of $\omega(v)$ as score(v), which is derived from two values, namely, the color number color(S) for $S \subseteq V(G)$ and the core number core(v) for $v \in V(G)$.

Definition 8 (*Color Number* color(S)) Given a graph G, a graph coloring $\mathcal{GC}$ is a mapping that maps each node $v \in G$ to a color (a number), such that no two adjacent nodes share the same color. Given a graph coloring $\mathcal{GC}$ for G, and a set of nodes $S \subseteq V(G)$, the color number of S, denoted by color$(S, \mathcal{GC})$, is the number of distinct colors in S. We use color(S) to denote color$(S, \mathcal{GC})$ if the context is self-evident.

Definition 9 (*Core Number* core(v)) Given a graph G and a node v, the core number of v, denoted by core(v), is the largest k s.t. there exists a subgraph $g \subseteq G$ with $v \in V(g)$, and for any node $u \in V(g)$, $d(u, g) \geq k$.

Computing the optimal graph coloring $\mathcal{GC}$ for graph G with minimum color$(V(G), \mathcal{GC})$ is an NP-hard problem [28]. Thus we adopt the Welsh-Powell algorithm [44], which uses a greedy strategy to compute a graph coloring $\mathcal{GC}$ in $O(m + n)$ time. The core number core(v) for all nodes $v \in V(G)$ can also be computed in $O(m + n)$ time [7]. For any $v \in V(G)$, we have the following fact.

Fact 1 $\min\{\text{core}(v), \text{color}(\{v \cup N(v)\})\} \geq \omega(v)$.

Based on Fact 1, we define score(v) as follows:

$$\text{score}(v) = \min\{\text{core}(v), \text{color}(\{v \cup N(v)\})\} \quad (1)$$

Given score(v) for all $v \in G$, the global pruning condition can be defined as follows.

Definition 10 (*Global Pruning Condition*) The global pruning condition GlobalPruning(v) used in line 5 of Algorithm 5 is defined as: score$(v) \leq |\text{priv}(C_{min})| + \alpha \times \frac{|\text{cov}(\mathcal{D})|}{|\mathcal{D}|}$.

Lemma 4 *The global pruning condition* GlobalPruning(v) *defined in Definition 10 is correct.*

Proof Sketch We only need to prove that once $\mathsf{GlobalPruning}(v)$ is satisfied for a certain v, even if we do not terminate the algorithm (line 5 of Algorithm 5), the candidate set $\mathcal{D}$ will not be updated. We prove this by contradiction. Note that once the condition $\mathsf{score}(v) \leq |\mathsf{priv}(C_{min})| + \alpha \times \frac{|\mathsf{cov}(\mathcal{D})|}{|\mathcal{D}|}$ is satisfied for a certain v, it will be satisfied for any of the remaining u (the node u that is processed after v in line 4 of Algorithm 5) if $\mathcal{D}$ is not updated, since $\mathsf{score}(u) \leq \mathsf{score}(v)$. Suppose $\mathcal{D}$ is updated to $\mathcal{D}'$ by replacing C_{min} with C when processing u with $\mathsf{score}(u) \leq \mathsf{score}(v)$, by the condition to update $\mathcal{D}$, we have $|\mathsf{priv}(C, \mathcal{D}')| > |\mathsf{priv}(C_{min})| + \alpha \times \frac{|\mathsf{cov}(\mathcal{D})|}{|\mathcal{D}|}$. However, by Fact 1, we have $|\mathsf{priv}(C, \mathcal{D}')| \leq \omega(u) \leq \mathsf{score}(u) \leq \mathsf{score}(v)$, and by the global pruning condition, we have $\mathsf{score}(v) \leq |\mathsf{priv}(C_{min})| + \alpha \times \frac{|\mathsf{cov}(\mathcal{D})|}{|\mathcal{D}|}$, which contradicts with $\mathsf{score}(v) > |\mathsf{priv}(C_{min})| + \alpha \times \frac{|\mathsf{cov}(\mathcal{D})|}{|\mathcal{D}|}$. □

Discussion. Note that the core numbers for all nodes in G and the graph coloring $\mathcal{GC}$ can both be computed in $O(m + n)$ time. Checking the global pruning condition for node v based on Definition 10 requires to traverse $N(v)$ only once to compute $\mathsf{color}(\{v \cup N(v)\})$. Therefore, the time complexity of EnumK (Algorithm 4) will not increase after applying global pruning.

4.3 Local pruning

In local pruning, we need to define a sufficient condition to stop expanding a partial clique R at early stages of Algorithm 4. We can make use of the information in the two sets P and R in the $\mathsf{CliqueAll}$ algorithm, where R is the current partial clique and P is the set of candidate nodes that can be used to expand R to maximal cliques. Intuitively, R can be pruned when $|P|$ is smaller enough, or most nodes in $P \cup R$ have been covered in the current $\mathcal{D}$.

Specifically, given the current candidate set $\mathcal{D}$, C_{min}, P, and R, we define four sets $P_a = P \setminus \mathsf{cov}(\mathcal{D})$, $P_b = P \cap \mathsf{priv}(C_{min})$, $R_a = R \setminus \mathsf{cov}(\mathcal{D})$, and $R_b = R \cap \mathsf{priv}(C_{min})$. The relationship of all the above sets is illustrated in Fig. 5a, b. The potential of the current partial clique R to be expanded to replace C_{min} depends on the size of the maximum clique in the set $P_a \cup P_b$, i.e., $\omega(P_a \cup P_b)$. However, similar to computing $\omega(v)$ for $v \in V(G)$, computing $\omega(S)$ for $S \subseteq V(S)$ is an NP-hard problem. Thus, instead of using $\omega(S)$ for any $S \subseteq V(S)$, we use an upper bound of $\omega(S)$, denoted by $\mathsf{score}(S)$, which is based on the following fact.

Fact 2 $\min\{\max_{v \in S} |N(v) \cap S|, \mathsf{color}(S)\} \geq \omega(S)$.

Based on Fact 2, we define $\mathsf{score}(S)$ as follows:

$$\mathsf{score}(S) = \min\{\max_{v \in S} |N(v) \cap S|, \mathsf{color}(S)\} \tag{2}$$

The local pruning condition is defined as follows.

Definition 11 (*Local Pruning Condition*) The local pruning condition $\mathsf{LocalPruning}(P, R)$ in line 11 of Algorithm 5 is: $\mathsf{score}(P_a \cup P_b) + |R_a \cup R_b| \leq |\mathsf{priv}(C_{min})| + \alpha \times \frac{|\mathsf{cov}(\mathcal{D})|}{|\mathcal{D}|}$.

Lemma 5 *The local pruning condition* $\mathsf{LocalPruning}(P, R)$ *defined in Definition 11 is correct.*

Proof Sketch We prove that once $\mathsf{LocalPruning}(P, R)$ is satisfied, if we continue expanding R, the candidate set $\mathcal{D}$ will not be updated. We prove this by contradiction. Suppose $\mathcal{D}$ is updated to $\mathcal{D}'$ by replacing C_{min} with $C = R \cup P'$ when expanding R for $P' \subseteq P$, and let $P'_a = P' \setminus \mathsf{cov}(\mathcal{D})$ and $P'_b = P' \cap \mathsf{priv}(C_{min})$, the relationship of R, P', P'_a and P'_b is illustrated in Fig. 5c. By the condition to update $\mathcal{D}$, we have $|\mathsf{priv}(C, \mathcal{D}')| > |\mathsf{priv}(C_{min})| + \alpha \times \frac{|\mathsf{cov}(\mathcal{D})|}{|\mathcal{D}|}$. However, by Fact 2 and the local pruning condition, we have $|\mathsf{priv}(C, \mathcal{D}')| = |P'_a \cup P'_b \cup R_a \cup R_b| \leq \omega(P'_a \cup P'_b) + |R_a \cup R_b| \leq \mathsf{score}(P'_a \cup P'_b) + |R_a \cup R_b| \leq |\mathsf{priv}(C_{min})| + \alpha \times \frac{|\mathsf{cov}(\mathcal{D})|}{|\mathcal{D}|}$, which contradicts with $|\mathsf{priv}(C, \mathcal{D}')| > |\mathsf{priv}(C_{min})| + \alpha \times \frac{|\mathsf{cov}(\mathcal{D})|}{|\mathcal{D}|}$. □

Discussion. In local pruning condition, we need to compute $\max_{v \in P_a \cup P_b} |N(v) \cap (P_a \cup P_b)|$ and $\mathsf{color}(P_a \cup P_b)$. Note that after checking $\mathsf{LocalPruning}(P, R)$ (line 11 of Algorithm 5), we need to compute $u \leftarrow \mathrm{argmax}_{v \in P \cup X}\{|P \cap N(v)|\}$ (line 12 of Algorithm 5). Obviously, the cost of computing u is no less than the cost of computing $\max_{v \in P_a \cup P_b} |N(v) \cap (P_a \cup P_b)|$ and $\mathsf{color}(P_a \cup P_b)$. Therefore, the time complexity of EnumK (Algorithm 4) will not increase after applying local pruning.

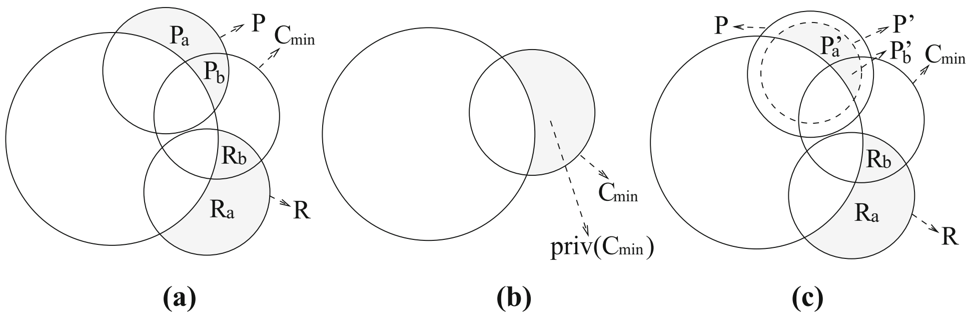

Fig. 5 Illustration for local pruning. **a** P and R, **b** cov(D), **c** P′ and R

Algorithm 6 InitK(graph $G = (V, E)$, integer k)

```
1: $\mathcal{S} \leftarrow \emptyset$; $U \leftarrow \emptyset$;
2: for all $v \in V$ in non-increasing order of score$(v)$ do
3:   if $|\mathcal{S}| = \eta \times k$ then break;
4:   if $v \notin U$ then
5:     $C \leftarrow$ CliqueGreedy$(N(v), \{v\})$;
6:     $\mathcal{S} \leftarrow \mathcal{S} \cup \{C\}$;
7:     for all $v \in C$ do $U \leftarrow U \cup \{v\} \cup N(v)$;
8: return top-$k$ cliques in $\mathcal{S}$ with maximum size;
9: procedure CliqueGreedy$(P, R)$
10: if $P = \emptyset$ then return $R$;
11: $u \leftarrow \text{argmax}_{v \in P}\{\min\{|P \cap N(v)|, \text{score}(v)\}\}$;
12: return CliqueGreedy$(P \cap N(u), R \cup \{u\})$;
```

4.4 Initial candidate computation

Recall that a better initial candidate clique set can potentially help both global pruning and local pruning to gain higher pruning power. In this subsection, we introduce a greedy algorithm to compute an initial candidate clique set $\mathcal{D}$. Intuitively, in a good $\mathcal{D}$, the size $|C|$ of each $C \in \mathcal{D}$ should be large, and the size $|C_i \cap C_j|$ for each pair $C_i \in \mathcal{D}$ and $C_j \in \mathcal{D}$ should be small.

Our algorithm InitK to compute the initial $\mathcal{D}$ is shown in Algorithm 6. Generally speaking, $\mathcal{D}$ is computed by generating a set $\mathcal{S}$ of $\eta \times k$ ($\eta \geq 1$) maximal cliques such that each $C \in \mathcal{S}$ is large, and for any two maximal cliques $C_i, C_j \in \mathcal{S}$, $C_i \cap C_j = \emptyset$. In other words, we generate a set of non-overlapping maximal cliques and select the largest k of them to be $\mathcal{D}$. In order to do this, we use U to maintain the set of nodes that are covered by cliques in $\mathcal{S}$ as well as their neighbors in G, i.e., $U = \bigcup_{C \in \mathcal{S}, v \in C}\{v\} \cup N(v)$. Both $\mathcal{S}$ and U are initialized to be $\emptyset$ (line 1). Recall that in Sect. 4.2, we show that for each node $v \in V(G)$, the potential size of the maximum clique containing v can be computed using score(v) (Eq. 1). Thus, in order to find large maximal cliques, we traverse $v \in V$ in non-decreasing order of score(v) (line 2) and stops the traversal whenever $\eta \times k$ maximal cliques are generated in $\mathcal{S}$ (line 3). In order to avoid overlapping, we compute a maximal clique from each v only if $v \notin U$ (line 4). The non-overlapping condition, i.e., the maximal clique generated from v does not overlap with any maximal cliques in $\mathcal{S}$, is guaranteed because by the definition of U and the condition $v \notin U$, we can guarantee that v is not covered by the current $\mathcal{S}$, and none of the neighbors of v is covered by $\mathcal{S}$. For each such a v, we use a greedy algorithm CliqueGreedy to compute a maximal clique C containing v (line 5), add C into $\mathcal{S}$ (line 6), and maintain U by adding C along with all neighbors of nodes $v \in C$ (line 7). Finally, after $\mathcal{S}$ is generated, we return the top-k cliques in $\mathcal{S}$ with the maximum size as the initial candidate maximal clique set (line 8). Next, we introduce how the greedy algorithm CliqueGreedy works to generate a potentially large maximal clique containing v.

Procedure CliqueGreedy(P, R). The algorithm CliqueGreedy (line 9–12 of Algorithm 6) adopts a recursive approach to generate a maximal clique where R is the current partial clique generated and P is the set of candidate nodes that can be added to R to form larger cliques. The algorithm stops when P is empty (line 10). Otherwise, it selects a node u from P that is likely to form the largest clique with nodes in P. Similar to global pruning (Sect. 4.2), the potential size of such maximum clique in P containing node v can be calculated as $\min\{|P \cap N(v)|, \text{score}(v)\}$ which is obviously an upper bound of the size of the maximum clique formed by v and other nodes in P. Therefore, u can be computed by selecting a node v in P that can maximize the potential size $\min\{|P \cap N(v)|, \text{score}(v)\}$ (line 11). After selecting u, we recursively invoke CliqueGreedy by adding the new selected node u into R, and updating P to be the set of nodes in the original P which are adjacent to u (line 12).

Discussion. Compared to CliqueAll used in EnumK (Algorithm 4) which enumerates all maximal cliques by expanding from every node $v \in V(G)$, in InitK (Algorithm 6) used in EnumKOpt (Algorithm 5), for each node $v \in V(G)$, we only generate one maximal clique. Obviously, the time cost of InitK is not larger than that of CliqueAll. Therefore, the time complexity of EnumK (Algorithm 4) will not increase after applying initial candidate computation. In summary, applying the three optimization strategies in EnumKOpt does not increase the time complexity of the EnumK algorithm.

5 An I/O efficient algorithm

Due to the rapid graph growth in the big data era, the size of many graphs increases sharply so that they cannot entirely reside in main memory. Motivated by this, in this section, we study a new I/O efficient algorithm for diversified top-k clique search in a graph G that cannot be entirely held in main memory.

5.1 A naive solution

A naive solution to this problem is that we maintain the candidate set $\mathcal{D}$ in memory and adopt an existing I/O efficient maximal clique enumeration algorithm such as the algorithm in [16] to enumerate all the cliques in the input graph to update $\mathcal{D}$. The naive solution SeqEnumK is shown in Algorithm 7.

SeqEnumK first initializes the top-k candidate set $\mathcal{D}$ (line 1). Then it repeatedly extracts a subgraph G_{S^+} (named extended subgraph in the paper) of G which can fit in memory (line 5) and computes the maximal cliques locally in the subgraph G_{S^+} (line 7). Here, S^+ is the union of nodes in S and their neighbors in G, and G_{S^+} is the subgraph induced by nodes in S^+. Line 5 is used to estimate whether the graph induced by S^+ can fit in main memory with size M by check-

Algorithm 7 SeqEnumK(graph $G = (V, E)$, integer k)

```
1: line 1-2 of Algorithm 4;
2: $\mathcal{S} \leftarrow \emptyset$; $\mathcal{S}^+ \leftarrow \emptyset$;
3: for each $v \in V$ in increasing order of $id(v)$ do
4:     $\mathcal{S} \leftarrow \mathcal{S} \cup \{v\}$; $\mathcal{S}^+ \leftarrow \mathcal{S}^+ \cup \{v\} \cup N(v, G)$;
5:     if ($\phi_{deg} \cdot |S^+| \geq cM$) then
6:         scan $G$ on disk once to extract $G_{S^+}$;
7:         apply CliqueAll in Algorithm 1 to compute $\mathcal{M}(G_{S^+})$;
8:         for each $C \in \mathcal{M}(G_{S^+})$ do
9:             if ($C \cap S \neq \emptyset$ and ($\min_{v \in C}\{id(v)\} \geq \min_{u \in S}\{id(u)\}$)) then
10:                Invoke CandMaintain ($C$) in Algorithm 4 to update $\mathcal{D}$;
11:        $\mathcal{S} \leftarrow \emptyset$; $\mathcal{S}^+ \leftarrow \emptyset$;
12: return $\mathcal{D}$;
```

ing the condition $\phi_{deg} \cdot |S^+| \geq cM$. The technique to compute ϕ_{deg} is introduced in details in [16]. For each computed maximal clique C, procedure CandMaintain is invoked to update the candidate set $\mathcal{D}$ (line 10). To avoid enumerating duplicate maximal cliques, SeqEnumK does not output a maximal clique that has been computed before. This is done by checking whether the node with the smallest id in C is greater than or equal to the node with the smallest id in the set S (line 9). The interested reader is referred to [16] for details.

SeqEnumK can be used to solve the top-k diversified clique search problem and can keep the same approximation ratio as our in-memory algorithm (Algorithm 5). However, this approach has the following drawbacks: (1) As stated in Sect. 1, the number of maximal cliques in a graph G can be exponential in the number of nodes in G, thus it is computationally intractable to enumerate all the maximal cliques in G when G is very large. (2) The graph partition strategy used in SeqEnumK leads to many duplicate maximal cliques during the computation, thus SeqEnumK needs to verify the duplicates in line 9, which results in the inefficiency of the algorithm. (3) In the SeqEnumK, optimization strategies are difficult to be adopted, especially for the global pruning and initial candidate computation strategies, because the maximal cliques can be generated in an arbitrary order.

5.2 A new approach

In this paper, we propose a new algorithm for the diversified top-k clique search problem in a massive graph, which can overcome the shortcomings of the naive approach while keeping the same worst-case approximation ratio. In the new algorithm, we process each node in an order based on their core number. As shown in Fact 1, for a node v, core number $\mathsf{core}(v)$ is an upper bound of $\omega(v)$, thus it can be used as a pruning indicator. With this processing order, we can prune unpromising clique enumeration at an early stage. Moreover, in order to reap the benefit of initial candidate computation, in our new algorithm we leverage the edges that have a large core number to compute the initial candidate clique set $\mathcal{D}$. In addition, instead of the extended subgraph used in SeqEnumK, we introduce a new partition paradigm using oriented subgraph as the basic component of the new algorithm. Compared with the extended subgraph, oriented subgraph does not need to verify the duplicates, which can further reduce unnecessary computation. By adopting these strategies, the optimization strategies can be applied naturally and effectively and the unnecessary computation can be largely reduced in our new algorithm. Before showing the details of our algorithm, we first introduce some definitions.

Definition 12 (*Seed Nodes/Subgraph*) Given a graph $G = (V, E)$, seed nodes, denoted by S, are a set of nodes selected from V. The seed subgraph, denoted by $G_S = (V_S, E_S)$, is the induced subgraph of G by S.

Definition 13 (*Node Order*). We define a total order $\prec$ for nodes in G as: given two node u and v, $u \prec v$ if and only if either of the following two conditions is satisfied:

1. $d(u) > d(v)$;
2. $d(u) = d(v)$ and $id(u) < id(v)$.

Definition 14 (*Oriented Nodes/Subgraph*). Given a set of seed nodes S and a graph $G = (V, E)$, the oriented nodes, denoted by S^*, is defined as $S^* = S \cup \{v : v \in N(u, G), u \in S, v \prec u\}$. The oriented subgraph, denoted by $G_{S^*} = (V_{S^*}, E_{S^*})$, is the induced subgraph of G by S^*.

When S contains only one node v, we call the corresponding oriented subgraph as a node-based oriented graph, denoted by G_{v^*}.

Definition 15 (*Core Number* $\mathsf{core}(e)$) Given a graph G and an edge $e = (u, v)$, the core number of an edge e, denoted by $\mathsf{core}(e)$, is defined as $\mathsf{core}(e) = \min\{\mathsf{core}(u), \mathsf{core}(v)\}$.

As oriented subgraph is used as the basic component in our new algorithm, we first prove that all the maximal cliques in the input graph can be computed locally from the oriented subgraphs. This property leads to the design of our oriented subgraph partition-based algorithm, which will be shown in Algorithm 8.

Lemma 6 *Given a graph $G = (V, E)$, let $S = \{S_1, S_2, ..., S_t\}$, where $\bigcup_{1 \leq i \leq t} S_i = V$ and $S_i \cap S_j = \emptyset$ for $i \neq j$, let $\mathcal{M}(G_{S_i^*})$ be the set of maximal cliques in $G_{S_i^*}$. For each maximal clique C in G, there is one and only one oriented subgraph $G_{S_i^*}$ such that $C \in \mathcal{M}(G_{S_i^*})$.*

Proof Sketch We first prove that for each maximal clique C, there exists one oriented subgraph $G_{S_i^*}$ such that $C \in \mathcal{M}(G_{S_i^*})$. Suppose that there is a maximal clique C' in G and there is no $S_i \in S$ such that $C' \in \mathcal{M}(G_{S_i^*})$. Then we can construct a seed set $S_{i'}$ with only one node v such that $S_{i'} = \{v : v \in C', u \prec v \text{ for any } u \in C' \backslash \{v\}\}$. It is obvious that $C' \in \mathcal{M}(G_{S_{i'}^*})$ and $v \notin S_i$ for any $1 \leq i \leq t$, which

Algorithm 8 IOEnumK(graph $G = (V, E)$, integer k)

```
1: line 1-2 of Algorithm 4;
2: compute the core number for each node using the I/O efficient algorithm in [14];
3: $(\mathcal{D}, G') \leftarrow$ IOInitK$(G, k)$;
4: $\mathcal{S} \leftarrow \emptyset$; $\mathcal{S}^* \leftarrow \emptyset$;
5: for each $v \in V'$ in non-increasing order of core$(v)$ do
6:   if $|\mathcal{D}| = k$ and GlobalPruning$(v)$ then break;
7:   $\mathcal{S} \leftarrow \mathcal{S} \cup \{v\}$; $\mathcal{S}' \leftarrow \emptyset$; load $N(u, G')$ into memory;
8:   for each $u \in N(v, G')$ do
9:     if GlobalPruning$(u)$ = false and $u \prec v$ then
10:      $\mathcal{S}' \leftarrow \mathcal{S}' \cup \{u\}$;
11:  $\mathcal{S}^* \leftarrow \mathcal{S}^* \cup \{v\} \cup \mathcal{S}'$;
12:  if $(\phi_{deg} \cdot |S^*| \geq cM)$ then
13:    scan $G'$ on disk once to extract $G'_{S^*}$;
14:    apply CliqueK to $G'_{S^*}$;
15:    $\mathcal{S} \leftarrow \emptyset$; $\mathcal{S}^* \leftarrow \emptyset$;
16: return $\mathcal{D}$;
17: procedure IOInitK$(G, k)$
18:   sort the edges based $e \in E(G)$ in non-increasing order of core$(e)$ on disk;
19:   load as many edges as possible with maximum core numbers to form subgraph $G_{init}$ into memory;
20:   apply Algorithm 5 to $G_{init}$ and obtain initial $\mathcal{D}$;
21:   extract subgraph $G'$ on disk consisting of edges $e$ with core$(e) >$ $|$priv$(C_{min})| + \alpha \times \frac{|\text{cov}(\mathcal{D})|}{|\mathcal{D}|}$;
22:   return $\mathcal{D}$ and $G'$;
```

contradicts that $\bigcup_{1 \leq i \leq t} S_i = V$. Thus, for each maximal clique C, there exists at least one oriented subgraph $G_{S_i^*}$ such that $C \in \mathcal{M}(G_{S_i^*})$.

We then prove that there is only one $G_{S_i^*}$ such that $C \in \mathcal{M}(G_{S_i^*})$. Suppose that there exists one maximal clique C' which is contained in $\mathcal{M}(G_{S_i^*})$ and $\mathcal{M}(G_{S_j^*})$. Then for the node v with the least order in C', it is obvious that $v \in S_i$ and $v \in S_j$, which contradicts that $S_i \cap S_j = \emptyset$ for $i \neq j$. Thus, for each maximal clique C in G, there is only one oriented subgraph $G_{S_i^*}$ such that $C \in \mathcal{M}(G_{S_i^*})$. □

Note that the maximal cliques computed locally from G_{S^*} may not be a maximal clique globally. However, using the oriented subgraph as the basic partition component for the diversified top-k clique search problem is reasonable as follows: for the application scenarios of diversified top-k clique search, the coverage of returned result is the most important focus of the applications. In our algorithm, we can guarantee that every maximal clique in the graph is enumerated in one oriented subgraph, which is proved in Lemma 6. In addition as shown in Lemma 7, our algorithm can still achieve a guaranteed approximation ratio of 0.25, which is the same as our in-memory algorithm (Algorithm 5). Thus, it is reasonable to adopt oriented subgraphs in our algorithm.

Algorithm IOEnumK Our new algorithm IOEnumK is illustrated in Algorithm 8. The framework of IOEnumK is similar to SeqEnumK. However, compared to SeqEnumK, in IOEnumK, we need to overcome the challenge to integrate the optimization strategies in the algorithm to reduce unnecessary computation. To solve this problem, IOEnumK contains two steps: (1) We first compute an initial candidate set $\mathcal{D}$ (procedure IOInitK) and prune the edges with core number smaller than $|\text{priv}(C_{min})| + \alpha \times \frac{|\text{cov}(\mathcal{D})|}{|\mathcal{D}|}$. The remaining edges form a subgraph G'. (2) We then enumerate all the maximal cliques in G' to update the candidate set $\mathcal{D}$ with pruning strategies (line 4–16). Such a procedure runs as follows: After computing the initial candidate set (line 3), we partition the input graph into smaller oriented subgraphs G_{S^*} each of which can be held in main memory (line 8-13) and compute the maximal cliques locally using CliqueK (line 14). To apply the GlobalPruning strategy to our algorithm, we compute the core number of nodes in line 2. Note that compared with EnumKOpt, IOEnumK just uses core(v) as score(v). The reason to adopt this strategy is that color(S) is hard to handle when the input graph cannot fit into memory. IOEnumK utilizes the I/O efficient core decomposition algorithm in [14] to compute the core number of nodes. To estimate the size of G_{S^*}, we use the similar method introduced in [16], i.e., $\phi_{deg} = \text{argmax}_{v \in S^*}\{deg(v)\}$ (line 12). In this paper, we assume that at least G_{v^*} can be held in memory for any $v \in V(G)$. This assumption is reasonable in practice. For example, in the datasets *UK-2005* and *Webbase* used in our experiment, the maximum G_{v^*} contains 1, 755, 518 and 1, 116, 179 edges, which covers only 0.19 and 0.11 % of the edges of the entire graph, respectively. After all the nodes are processed, IOEnumK returns the top-k results in line 16.

Procedure IOInitK For the IOInitK procedure, we aim to compute k initial candidates to improve the pruning power. In this procedure, we first construct a subgraph G_{init} by loading as many edges as possible in non-increasing order of their core numbers and then utilize the in-memory algorithm EnumKOpt (Algorithm 5) to obtain the initial candidates. The reasons to utilize G_{init} here are as follows: (1) Compared with oriented subgraph, the subgraph consisting of edges with large core numbers is usually not huge and we can avoid estimating its size, thus the limited main memory can be utilized more effectively. (2) The I/O efficient algorithm for core decomposition has been explored in literature; hence, the subgraph can be constructed easily. (3) core(v) will be used in the following processing steps, thus we can share the core number information with low extra cost. IOInitK first sorts the edges of G based on the core(e) and extracts the subgraph G_{init} by loading as many edges as possible in non-increasing order of their core numbers(line 18-19), then the in-memory EnumKOpt(Algorithm 5) is invoked to obtain the initial candidates (line 20). Note that cliques chosen in line 20 are the cliques with size larger than the smallest core number of edges loaded in main memory. This is because only the cliques with size larger than the smallest core number of edges loaded in main memory can be guaranteed to be maximal in the original input graph. After obtaining the initial $\mathcal{D}$, IOInitK prunes the edges which cannot be used to further improve the candidate set $\mathcal{D}$ and extract G' (line 21) for further refining $\mathcal{D}$. G' is stored in adjacency lists and the nodes are ordered by their core number $\mathsf{core}(v)$. G' can be easily computed in $O(sort(m))$ I/Os by sorting. For each node u, the degree information $d(u)$ is also embedded in the node u. In this way, the node order can be easily obtained in line 9.

The following Lemma 7 shows the top-k diversified cliques returned by IOEnumK can achieve the same guaranteed approximation ratio of 0.25 as EnumKOpt.

Lemma 7 *Given a graph G and an integer k, suppose $\mathcal{D}^*$ is the optimal diversified top-k cliques, and $\mathcal{D}$ is the result returned by* IOEnumK*, then* $|\mathsf{cov}(\mathcal{D})| \geq 0.25 \times |\mathsf{cov}(\mathcal{D}^*)|$

Proof Sketch The proof simply follows the proof of Lemma 1. As shown in the Lemma 6, all the maximal cliques in original input graph G are processed in IOEnumK, then IOEnumK has the same settings as EnumKOpt. Thus the lemma is proved directly. □

Algorithm Analysis. To analyze the complexity of IOEnumK, we use the standard I/O complexity notations [1] as follows: M is main memory size and B is the disk block size ($1 \ll B \ll M/2$). The I/O complexity to scan N elements is $scan(N) = \Theta(\frac{N}{B})$, and the I/O complexity to sort N elements is $sort(N) = O(\frac{N}{B} \cdot \log_{\frac{M}{B}} \frac{N}{B})$.

The cost of IOEnumK contains three parts: (1) compute the core number of nodes in the input graph. (2) sort the graph. (3) partition the graph and compute the maximal cliques. For part (1), as shown in [14], it takes $O(k_{max}(m+n))$ CPU time and $O(\frac{k_{max}(m+n)}{B})$ I/Os, where k_{max} is the maximum core number of the input graph. For part (2), as shown in [1], it takes $O(m\log(m))$ CPU time and $O(sort(m))$ I/Os. For part (3), IOEnumK needs to scan G' for $(s+1)$ times, where s is number of G_{S^*} computed in line 13. As $(\phi_{deg} \cdot |S^*|) \geq cM$ in line 12 and $|S^*| \leq (\phi_{deg} \cdot |S|)$, we have $|S| \geq \frac{cM}{(\phi_{deg})^2}$. As $c < 1$ is a constant, thus $s = O(\frac{n}{|S|}) = O(\frac{n \cdot \phi_{deg}^2}{M})$. Hence, for part (3), IOInitK takes $O(s \cdot T_{enum}(G_{S^*}))$ CPU time and $O(s \cdot scan(m+n))$ I/Os. Thus, the overall CPU time that IOEnumK requires is $O(k_{max}(m+n) + m\log(m) + s \cdot T_{enum}(G_{S^*}))$ and the overall I/Os is $O(\frac{k_{max}(m+n)}{B} + sort(m) + s \cdot scan(m+n))$.

Discussion. Note that the oriented subgraph G_{S^*} defined in Definition 14 is different from the extended subgraph G_{S^+} defined in [16], which leads to totally different algorithm design. In [16], the extended subgraph contains all edges among the nodes in $S \cup \{v : v \in N(u, G), u \in S\}$, while in our definition, we extend the subgraph by adding the neighbors v of $u \in S$ where $v \prec u$. Compared to the extended subgraph, oriented subgraph has the following advantages: (1) With respect to the same seed nodes S, oriented subgraph G_{S^*} is generally smaller than extended subgraph G_{S^+}, which means we can contain more seed nodes in one partition with the same memory and complete the algorithm with less scans. (2) As there is no duplicates among different oriented subgraphs, compared to SeqEnumK (line 9), it is not necessary to verify whether the computed maximal cliques have been generated in other subgraphs. Both of the advantages improve the performance of IOEnumK.

In [46], the authors propose a distributed algorithm for maximal clique enumeration and a node order is defined in their paper. The node order is used in both [46] and our paper, but in different ways. In Definition 14, the oriented subgraph is defined based on the node order and it aims to reduce the size of a partition to be loaded in memory. However, in [46], the node order is used to reduce the computational and communication cost, but it cannot be used to reduce the size of each partition. Specifically, in order to guarantee that the cliques generated are maximal cliques, in the algorithm proposed in [46], given a set of nodes S, a partition still needs to maintain the subgraph induced by the node set $S^+ = S \cup \{v : v \in N(u, G)\}$, which can be much larger than S^* used in our paper when S contains some high-degree nodes.

6 Related work

We review the related work to diversified top-k clique search problem from five categories, namely, maximal clique enumeration, maximum clique computation, max k-cover, diversified top-k search and I/O efficient graph algorithms.

Maximal Clique Enumeration. Maximal clique enumeration is a fundamental graph problem and has been extensively studied. Most algorithms for maximal clique enumeration (e.g., [11] and [3]) are based on backtracking search. Tomita et al. [40] and Eppstein and Strash [23] further speedup maximal clique enumeration by selecting good pivots to reduce the search path in backtracking. Maximal clique enumeration in a sparse graph is studied in [13]. A near-optimal algorithm for maximal clique enumeration in a sparse graph is given in [22]. Recently, Wang et al. [43] propose an algorithm to enumerate maximal cliques by taking the overlaps among cliques into consideration. Both [22] and [43] are introduced in details in Sects. 2.3 and 2.4, respectively. In addition, parallel maximal clique enumeration is studied in [37], and I/O efficient maximal clique enumeration algorithms are proposed in [15] and [16]. A distributed algorithm for maximal clique enumeration is proposed in [46].

Maximum Clique Computation. The maximum clique within a graph G is the largest subgraph of G that is a clique. A classical algorithm for maximum clique computation is proposed in [12]. At every stage of the algorithm, it maintains the largest known clique size ω. For each subgraph G', the algorithm finds the set of nodes $\{w\}$ which are not in G' but are connected to all nodes in G'. Let m be the number of nodes in $\{w\}$. If $m + |G'| \leq \omega$, the subgraph G' is pruned. [33] introduces an additional pruning strategy based on vertex order. In [29,38,39], graph coloring is used to obtain an upper bound of the size of maximum clique to further prune the unnecessary computation. A distributed algorithm based on MapReduce for maximum clique computation is proposed in [45]. Based on maximum clique computation, a naive solution to diversified top-k clique search problem runs as follows: we first find the maximum clique C in G and remove the nodes and edges in C from G and repeat this procedure k times. The obtained k maximum cliques are returned as the result. However, this approach is not scalable as it needs to scan the graph k times. We evaluate this approach in our experiment (Exp-1 in Sect. 7).

Max k-Cover. As shown in Sect. 2.3, the greedy algorithm to compute the max k-cover can achieve an approximation ratio of $(1 - 1/e)$ which cannot be improved by any polynomial time algorithm unless P = NP [25]. Some other works focus on computing max k-cover in a streaming environment [5,6,36,47]. In this paper, as shown in Sect. 3.1, we generalize the algorithm introduced in [5] which is an improvement of the algorithm introduced in [36]. In [47], an algorithm is designed in a way that a new set is retained if it has the potential to cover some new nodes in the graph, and an existing set that does not cover any new nodes is removed. After processing all sets, the k retained sets with largest size are returned. In [6], the algorithm maintains multiple lists of sets $\mathcal{D}_{\delta_1}, \mathcal{D}_{\delta_2}, \ldots$ for $1 < \delta_1 < \delta_2 < \ldots$. Suppose $d_i = (\delta_i/2 - |\mathsf{cov}(\mathcal{D}_{\delta_i})|)/(k - |\mathcal{D}_{\delta_i}|)$, for a new set C, it is inserted into $\mathcal{D}_{\delta_i}$ only if C can cover at least d_i new nodes in $\mathcal{D}_{\delta_i}$. After processing all sets, the $\mathcal{D}_{\delta_i}$ that covers most nodes is returned. The approaches in [47] and [6] have better approximation ratio than [5] and [36] theoretically; however, neither of them can lead to efficient pruning strategies for diversified top-k clique search, thus they are not suitable to handle very large graphs as confirmed in our experiments. Max k-cover computation in MapReduce is studied in [17].

Diversified Top-k Search. Diversified top-k search, which aims at computing the top-k answers that are most relevant to a user query by taking diversity into consideration, has been extensively studied. In the literature, many existing solutions focus on answering the diversified top-k query for a specific problem. For example, diversified top-k document retrieval is studied by Agrawal et al. [2] and Angel and Koudas [4]. Lin et al. [31] study the k most representative skyline problem. Demidova et al. [19] study the diversified keyword query interpretation over structured databases. Diversified top-k graph pattern matching is studied by Fan et al. [24]. However, none of the above approaches can be used to efficiently compute diversified top-k cliques. A survey for different query result diversification approaches is provided by Drosou and Pitoura [21]. Some other works focus on a general framework for top-k answer diversification. For example, the general framework to answer diversified top-k queries is studied by Qin et al. [34] and Vieira et al. [41]. Top-k result diversification on a dynamic environment is studied by Minack et al. [32] and Borodin et al. [10]. The complexity of query result diversification is analyzed by Deng and Fan [20]. Nevertheless, the diversity of all the above frameworks is based on the pair-wise dissimilarity of query results, which is not applicable to the diversified top-k clique search problem studied in this paper.

I/O Efficient Graph Algorithms. Due to the rapid increase of graph size, traditional (in-memory) graph algorithms cannot be applied to handle large disk-resident graphs because of the I/O communication generated. Therefore, several graph algorithms focusing on I/O efficiency have been proposed in the literature. In [14], Cheng et al. devise a top-down approach for the core decomposition problem in massive networks. Triangle listing problem in massive graphs is studied in [18,27]. Cheng et al. also propose an I/O efficient algorithm for maximal clique enumeration problem by recursively extracting a core part of the input graph [15,16]. The I/O efficient algorithm for k-truss problem is investigated in [42]. The authors propose a bottom-up algorithm and a top-down algorithm to address k-truss decomposition problem in different application scenarios. Zhang et al. [50] study an I/O efficient semi-external algorithm to find all strongly connected components (SCC) in a graph, and they extend the algorithm in the external memory model when the nodes of graph cannot be kept in memory in [49]. Recently, an I/O efficient semi-external algorithm for depth first search problem is studied in [51].

7 Performance studies

In this section, we show our experimental results. All of our experiments are conducted on a machine with an Intel Xeon 3.4GHz CPU (8 cores) and 32GB main memory (for I/O efficient algorithm testing, we set the available memory as 1GB) running Linux (Red Had Enterprise version 6.4, 64bit).

Datasets. We use eight real-world large graphs with different types and graph properties (see Table 1) for testing. Among them, *Google*, *Skitter*, and *Pokec* are downloaded from SNAP (http://snap.stanford.edu), *Wiki* and *Youtube* are downloaded from KONECT (http://konect.uni-koblenz.de), *UK-2002*, *UK-2005* and *Webbase* are downloaded from WEB (http://law.di.unimi.it). *Google* was released in 2002 by Google as a part of Google Programming Contest. *Skitter* is an Internet topology graph released in 2005. *Pokec* is the most popular on-line social network in Slovakia. *Wiki* is the edit network of the Italian Wikipedia. *Youtube* is the social network of Youtube users and their connections. *UK-2002* and *UK-2005* are webpages crawled from .uk domain in 2002 and

Table 1 Datasets used in experiments

Dataset G	Type	$\lvert V(G)\rvert$	$\lvert E(G)\rvert$	Avg Degree
Google	Web	875,713	5,105,039	11.66
Skitter	Physical	1,696,415	11,095,298	13.08
Youtube	Social	3,223,589	12,223,774	7.58
Pokec	Social	1,632,803	30,622,564	37.51
Wiki	Reference	2,936,413	104,673,033	71.29
UK-2002	Web	18,520,486	298,113,762	32.19
UK-2005	Web	39,459,925	936,364,282	47.46
Webbase	Web	118,142,155	1,019,903,190	17.27

2005 respectively, in which nodes represent pages and edges represent hyperlinks between them. *Webbase* is obtained from the 2001 crawl perform by the WebBase crawler. We also evaluate the algorithms on synthetic graphs. The graph generator used in the tests is GTgraph (http://www.cse.psu.edu/~kxm85/software/GTgraph/).

Algorithms. We implement and compare twelve algorithms: the first ten are in-memory algorithms; SeqEnumK and IOEnumK are I/O efficient algorithms.

– EnumAll: Algorithm 1 (Sect. 2.3).
– EnumSub: Algorithm 2 (Sect. 2.4).
– EnumK: Algorithm 4 (Sect. 3.2).
– Local: EnumK + local pruning (Sect. 4.3).
– Global: Local + global pruning (Sect. 4.2).
– EnumKOpt: Global + InitK (Sect. 4.4).
– SOPS: Candidate maintenance using the method in [36].
– GOPS: Candidate maintenance using the method in [47].
– SIEVE: Candidate maintenance using the method in [6].
– MaxK: The algorithm based on maximum clique computation algorithm in [38] (Sect. 6).
– SeqEnumK: Algorithm 7 (Sect. 5.1).
– IOEnumK: Algorithm 8 (Sect. 5.2).

All algorithms are implemented in C++. For EnumAll, the source code is obtained from the author in [22]. For EnumSub, we download the source code from the homepage (http://appsrv.cse.cuhk.edu.hk/~jwang/) of the first author of [43]. We use the randomized algorithm RMCE in [43] for EnumSub and set τ as 0.8 which is the default setting in [43]. For SOPS, GOPS, and SIEVE, we apply all optimization techniques introduced in [36], [47], and [6] respectively, and also apply the early termination techniques introduced in Sect. 4 whenever possible. For SeqEnumK, we apply local pruning technique introduced in Sect. 4. For tests about in-memory algorithms on real graphs and synthetic graphs, we report the total processing time and the number of nodes covered by the top-k maximal cliques returned. For tests about I/O efficient algorithms, we also report the number of I/Os during the processing (as the number of covered nodes has the similar trend to that of the in-memory algorithms, the results are not shown in this part). We set the maximum running time for each test to be 10,000 seconds. If a test does not stop in the time limit, or fails due to out of memory exception, we denote the corresponding processing time as INF. For GOPS, we find out that it cannot terminate in the time limit for almost all tests due to its costly set update operation and low pruning power. Thus, we omit the result for GOPS in the experiments. Exp-(1-6) test the in-memory algorithms on real graphs, Exp-(7-9) test the in-memory algorithms on synthetic graphs and Exp-(10-11) test the I/O efficient algorithms.

Parameters. For tests on real graphs, we vary four parameters in our experiments, namely, k (the top-k value), α (the parameter used in procedure CandMaintain (refer to Algorithm 4)), η (the parameter used in procedure InitK (refer to Algorithm 6)), and $|V|$ (the graph size). k is selected from 10, 20, 30, 40, 50 with a default value of 40. α is selected from 0.1, 0.2, ..., 1 with a default value of 0.3. η is selected from 0, 1, 2, 3, 4, 5 with a default value of 3, where $\eta = 0$ means that no initial candidates are computed. For $|V|$, we generate subgraphs with 20, 40, 60, 80 %, and 100 % nodes of the original graph for each dataset, with a default value of 100 %. For tests on synthetic graphs, we vary $|V|$, k and the maximum clique size. $|V|$ is selected from $1M$, $1.5M$, $2M$, $2.5M$, $3M$ with a default vale of $2M$. The maximum clique (C_{max}) size is selected from 0.5, 0.75, 1, 1.25, 1.5‱ of $|V|$ with default value of 1‱. For I/O efficient algorithms, we vary k and α. Unless specified otherwise, when varying a certain parameter, the values of the other parameters are set to their default values.

7.1 In-memory algorithms on real graphs

Exp-1: Vary k (Efficiency). In this experiment, we vary k from 10 to 50. The curves of processing time for all the algorithms in the six datasets are shown in Fig. 6a–f respectively. For all algorithms, when k increases, the processing time increases. EnumAll and EnumSub perform worse than all the other algorithms in all datasets. This is because both EnumAll and EnumSub need to generate a large number of maximal cliques which is costly, and processing the greedy max k-cover algorithm on such a large number of maximal cliques is also costly. EnumAll outperforms EnumSub on *Google* (Fig. 6a) and *Youtube* (Fig. 6c), this is because EnumSub may spend extra cost to compute the sampling probability to obtain the summary. For *Pokec* (Fig. 6d) and *Skitter* (Fig. 6b), the efficiency of EnumSub is similar to or better than EnumAll when $k \geq 40$. SOPS is slower than other algorithms except for EnumAll and EnumSub in all datasets, because the maximal clique swapping operation in SOPS is costly, and early pruning has no large effect on SOPS. Among the other five algorithms, SIEVE is better

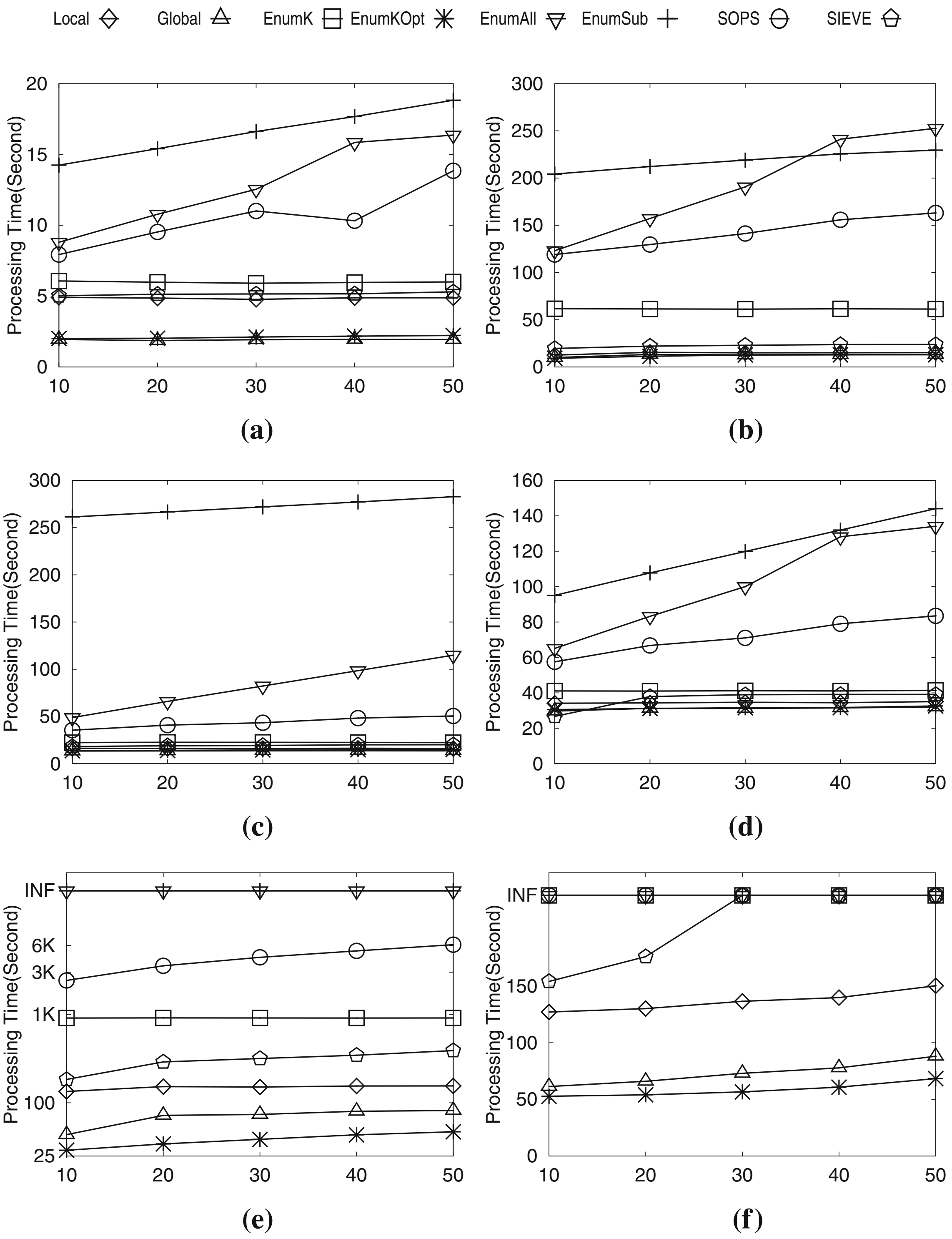

Fig. 6 Vary *k* (efficiency) **a** Google, **b** Skitter, **c** Youtube, **d** Pokec, **e** Wiki, **f** UK-2002

than EnumK in all datasets, because some early pruning techniques are applied on SIEVE, while EnumK has to enumerate all maximal cliques. However, after applying local pruning, the algorithm Local performs better than SIEVE in all datasets, which shows the high pruning power of the local pruning strategy used in Local. After applying global pruning, our algorithm Global improves Local by 30–300 % in terms of efficiency. And with initial candidate computation, EnumKOpt further improves that of Global. The pruning power of our three pruning strategies varies for different datasets. For *Google* (Fig. 6a), the global pruning strategy has the best pruning power. For *Wiki* (Fig. 6e), the prun-

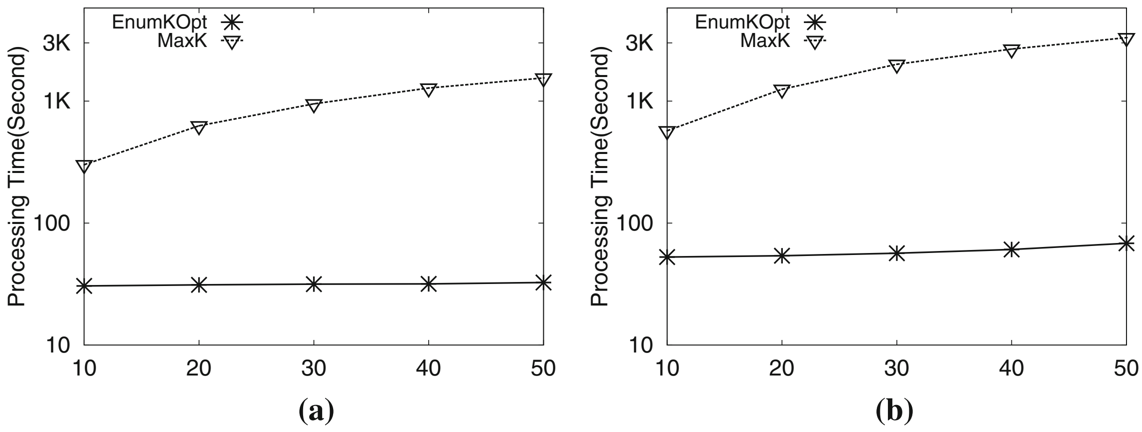

Fig. 7 Vary k (efficiency of MaxK). **a** Pokec, **b** UK-2002

ing power of the local pruning strategy is more evident. For *Youtube* (Fig. 6c), the pruning power of three pruning strategies is similar. In Fig. 6a–d, when the size of the dataset is small, the advantage of EnumKOpt is not obvious. However, in Fig. 6e, f, when the size of the dataset is large, EnumKOpt is much faster than all other algorithms. For example, in *Wiki* (Fig. 6e), EnumKOpt is an order of magnitude faster than SIEVE, and in *UK-2002* (Fig. 6f), when $k > 20$, SIEVE cannot terminate in the time limit, while EnumKOpt can finish in one minute for all k values. The results of MaxK on *Pokec* and *UK-2002* are shown in Fig. 7. When we vary k, the processing time of EnumKOpt keeps stable while that of MaxK increases linearly to k. This is because MaxK needs to search the graph k times and each time only one maximum clique is generated, whereas EnumKOpt only needs to search the graph once to obtain k cliques.

Exp-2: Vary k **(Effectiveness)**. In this experiment, we compare the number of nodes covered by the result returned from different algorithms. The results for the six datasets when varying k from 10 to 50 are shown in Fig. 8a–f, respectively. Since the results for EnumK, Local, and Global are the same, we only show the result for Global in Fig. 8. For MaxK, the results are the same as EnumAll. In general, when k increases, the number of covered nodes for all algorithms increases. For the four small datasets *Google*, *Skitter*, *Youtube*, and *Pokec* in Fig. 8a–d respectively, EnumAll performs best, followed by EnumSub and EnumKOpt. However, the number of covered nodes in EnumKOpt is very close to that in EnumAll, i.e., no less than 90 % of the number of covered nodes in EnumAll in most cases. The other three algorithms Global, SOPS, and SIEVE have similar performance which is worse than EnumKOpt in most cases. Such a result indicates that a good initial candidate set generated in EnumKOpt can improve both efficiency and effectiveness. For the large dataset *Wiki* (Fig. 8e), EnumSub and EnumAll cannot stop in the limited time and among the other four algorithms, our algorithm EnumKOpt performs best for all k values. For the dataset *UK-2002* (Fig. 8f) only our algorithms Global and EnumKOpt can terminate for all k values, SIEVE can only finish when $k \leq 20$. EnumKOpt performs best in all cases. In the following, for our proposed algorithms EnumK, Local, Global, and EnumKOpt, we only show the results for EnumKOpt, since their relative performances are similar to those shown in Figs. 6 and 8.

Exp-3: Vary α. We vary α from 0.1 to 1.0 and test both the efficiency and effectiveness of our algorithm EnumKOpt. The results for *Youtube*, *Pokec*, and *UK-2002* are shown in Fig. 9. In general, when α is smaller, the algorithm spends more time, but the corresponding result covers more nodes. For the small datasets *Youtube* (Fig. 9a, b) and *Pokec* (Fig. 9c, d), the efficiency is not as sensitive to α as the effectiveness. For the large dataset *UK-2002* (Fig. 9e, f), both efficiency and effectiveness are sensitive to α when α is small ($\leq$0.5), and not sensitive to α when α is large (>0.5). The results on the other three datasets are similar thus are omitted.

Exp-4: Vary η. We vary η from 0 to 5 and test the procedure InitK in EnumKOpt, where $\eta = 0$ indicates that no InitK is used. The results for efficiency and effectiveness are shown in Fig. 10a, b respectively. In general, when η increases from 0 to 3, the processing time on all datasets tends to decrease, while the number of covered nodes tends to increase. This is because that the pruning power of the global and local pruning strategies increases when η increases from 0 to 3. However, when η further increases, the processing time on all datasets tends to increase, while the number of covered nodes keeps unchanged. The reason is that when η further increases, the procedure InitK takes more time to compute the initial candidate set, while the pruning power of global and local pruning does not increase significantly.

Exp-5: Vary $|V|$. We vary $|V|$ from 20 to 100 % of the original graph and test the scalability of the algorithms on the two large datasets *Wiki* and *UK-2002*. The results are shown in Fig. 11. In general, when $|V|$ increases, both the

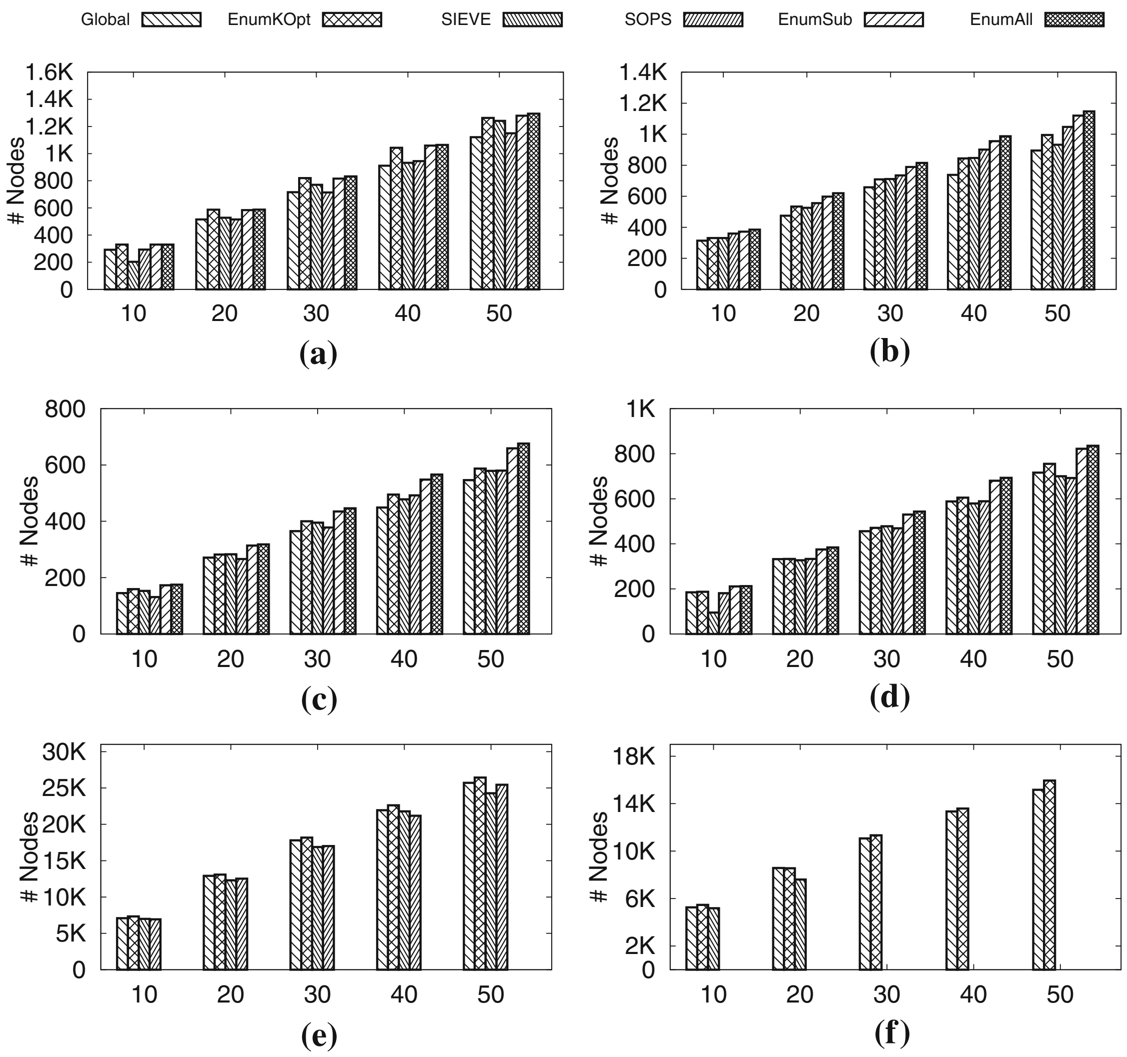

Fig. 8 Vary k (Effectiveness). **a** Google, **b** Skitter, **c** Youtube, **d** Pokec, **e** Wiki, **f** UK-2002

processing time and the number of covered nodes increase for all algorithms. EnumKOpt performs best in terms of both efficiency and effectiveness in all tests. Remarkably, in the *Wiki* dataset (Fig. 11a, b), the efficiency of EnumKOpt is much better than all other algorithms and the effectiveness of EnumKOpt is even better than that of EnumAll. For *UK-2002* (Fig. 11c, d), when $|V|$ increases, the processing time for EnumKOpt increases very stably while that for the other algorithms increases sharply. When $|V| > 60\,\%$, only EnumKOpt can finish in the time limit. Thus, EnumKOpt has high scalability.

Exp-6: PNP-Index**Test**. In this experiment, we test the PNP-Index size and the maintenance cost in different real datasets. All the parameters are set as the default values and the results are shown in Table 2. For the PNP-Index size, as the size of the graph increases, the index size also increases. But the size of PNP-Index is small compared with the size of the input graph. This is because only the top-k promising maximal cliques and five additional components are stored in the PNP-Index. Remarkably, for the large datasets *Wiki* and *UK-2002*, the size of PNP-Index is only 4.60 and 7.07 % of the size of the input graph, respectively. As the maintenance cost is always less than 0.01 % of the total processing time for all datasets, the results are not shown in Table 2.

7.2 In-memory algorithms on synthetic graphs

Exp-7: Vary $|V|$. We vary $|V|$ from $1M$ to $3M$ and test both the efficiency and effectiveness of the algorithms. The results are shown in Fig. 12. In general, when $|V|$ increases, both the processing time and the number of covered nodes increase for all algorithms. EnumKOpt performs best in terms of both efficiency and effectiveness in all tests. Remarkably, EnumSub performs well on synthetic graphs. It consumes less time than EnumAll, SOPS and SIEVE (Fig. 12a), while the number of covered nodes of it is the same as that of

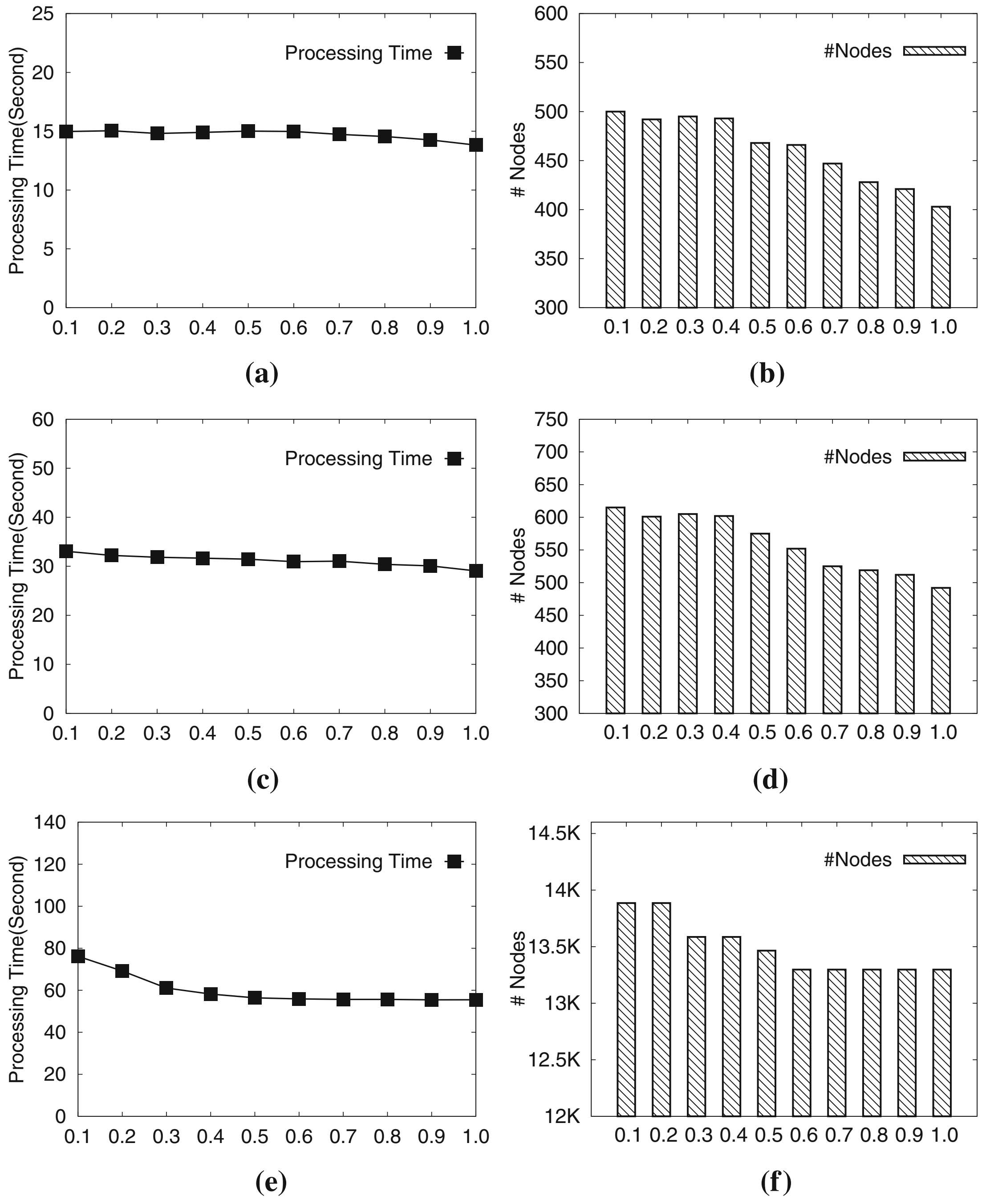

Fig. 9 Vary α in algorithm. **a** Youtube (Time), **b** Youtube (# covered nodes), **c** Pokec (Time), **d** Pokec (# covered nodes), **e** UK-2002 (Time), **f** UK-2002 (# covered nodes)EnumKOpt

EnumKOpt (Fig. 12b). SOPS performs worst in terms of effectiveness in all tests (Fig. 12b). When $|V| \geq 2.0M$, EnumAll can not finish the test. This is because too many maximal cliques are stored in memory for EnumAll during processing.

Exp-8: Vary k. We vary k from 10 to 50 and test both the efficiency and effectiveness of the algorithms. The results are shown in Fig. 13. Generally, when k increases, the number of covered nodes increases for all algorithms. Different from the results on real graphs, the processing time of all the algorithms keeps stable as k increases. EnumKOpt performs best, followed by EnumSub, in terms of both efficiency and effectiveness in all tests. SIEVE consumes less time than SOPS (Fig. 13a) and performs better than SOPS in terms of effectiveness (Fig. 13b). SOPS has the least number of covered nodes in all tests (Fig. 13b). EnumAll cannot finish

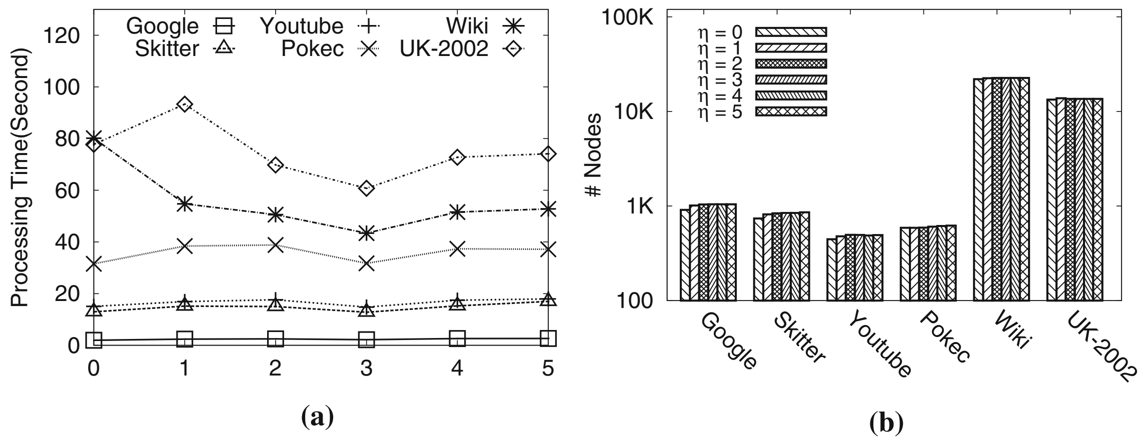

Fig. 10 Vary η in algorithm. **a** Time, **b** # covered nodes EnumKOpt

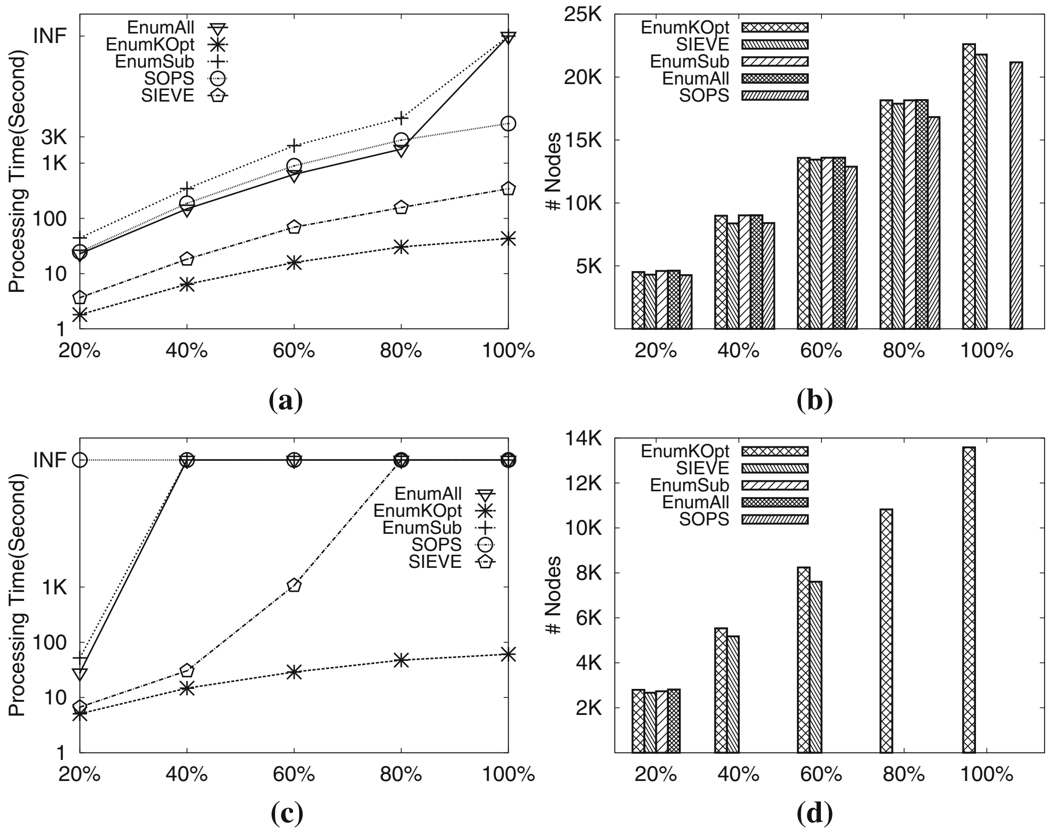

Fig. 11 Vary $|V|$ (scalability). **a** Wiki (Time), **b** Wiki (# covered nodes), **c** UK-2002 (Time), **d** UK-2002 (# covered nodes)

Table 2 PNP-Index size

Dataset	*Google*	*Skitter*	*Youtube*	*Pokec*	*Wiki*	*UK-2002*
Index size (MB)	7.99	15.48	42.31	14.89	38.90	168.69
Index/graph (%)	19.56	17.44	43.27	6.07	4.60	7.07

within the resource limit in all tests. The reason is that maintaining all the generated cliques consumes too much memory for EnumAll.

Exp-9: Vary $|C_{max}|$. We vary $|C_{max}|$ from 0.5 to 1.5‱ of the default value of $|V|$ and test both the efficiency and effectiveness of algorithms. The results are shown in Fig. 14

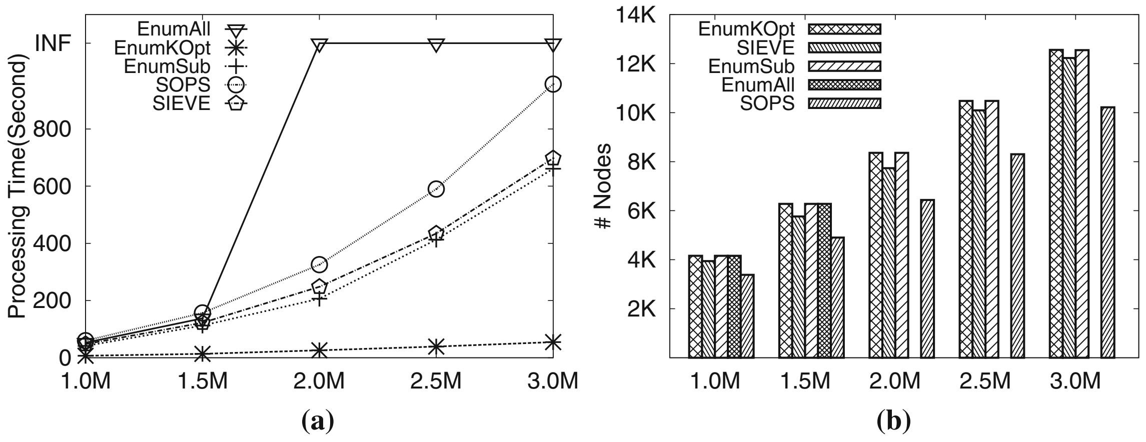

Fig. 12 Vary $|V|$ (synthetic graph). **a** Time, (b) # covered nodes

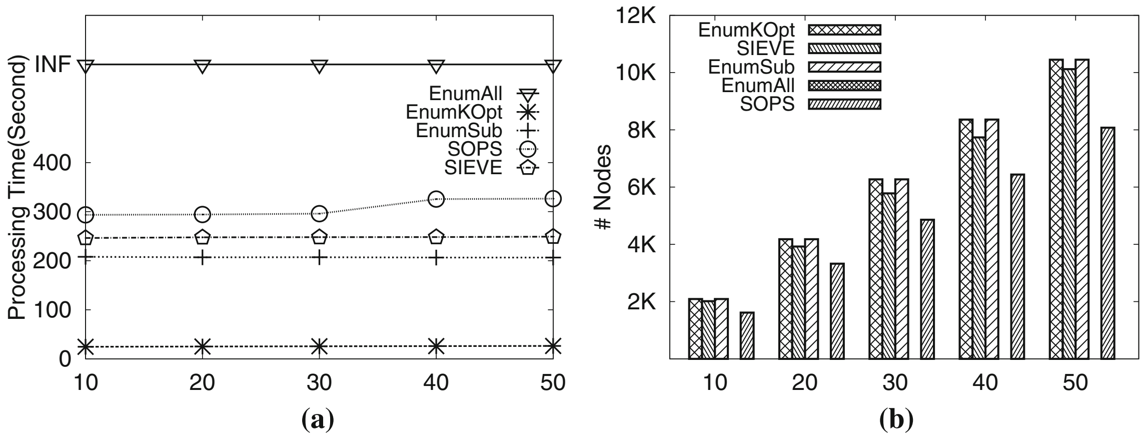

Fig. 13 Vary k (synthetic graph). **a** Time, (b) # covered nodes

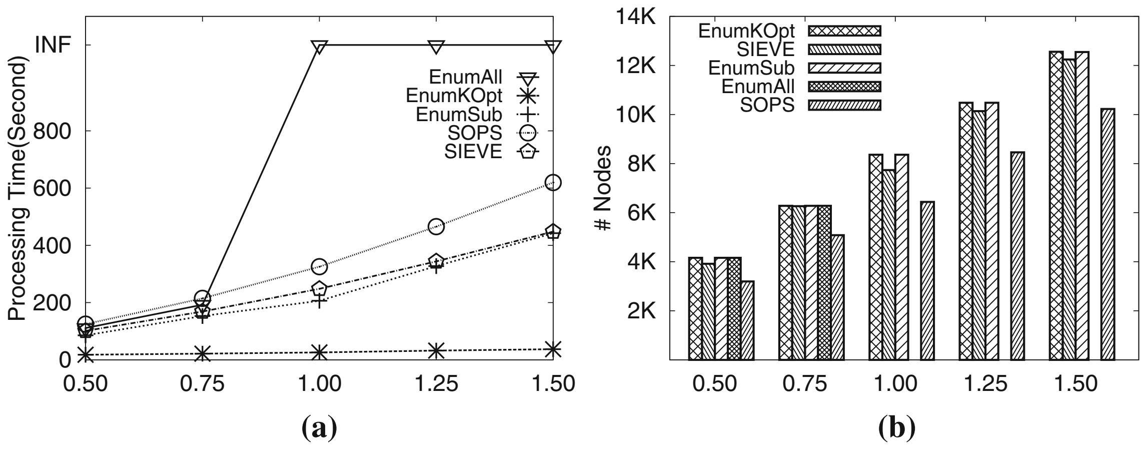

Fig. 14 Vary $|C_{max}|$ (synthetic graph). **a** Time, (b) # covered nodes

(The scale of x axis is ‱). In general, when $|C_{max}|$ increases, both the processing time and the number of covered nodes increase for all algorithms. EnumKOpt performs best, followed by EnumSub, in terms of both efficiency and effectiveness in all tests. SIEVE consumes similar time to EnumSub (Fig. 14a) for all tests but its number of covered nodes is less than that of EnumSub (Fig. 14b). SOPS performs worst in terms of effectiveness in all tests (Fig. 14b). When $|C_{max}| \geq 1‱$, EnumAll can not finish the test.

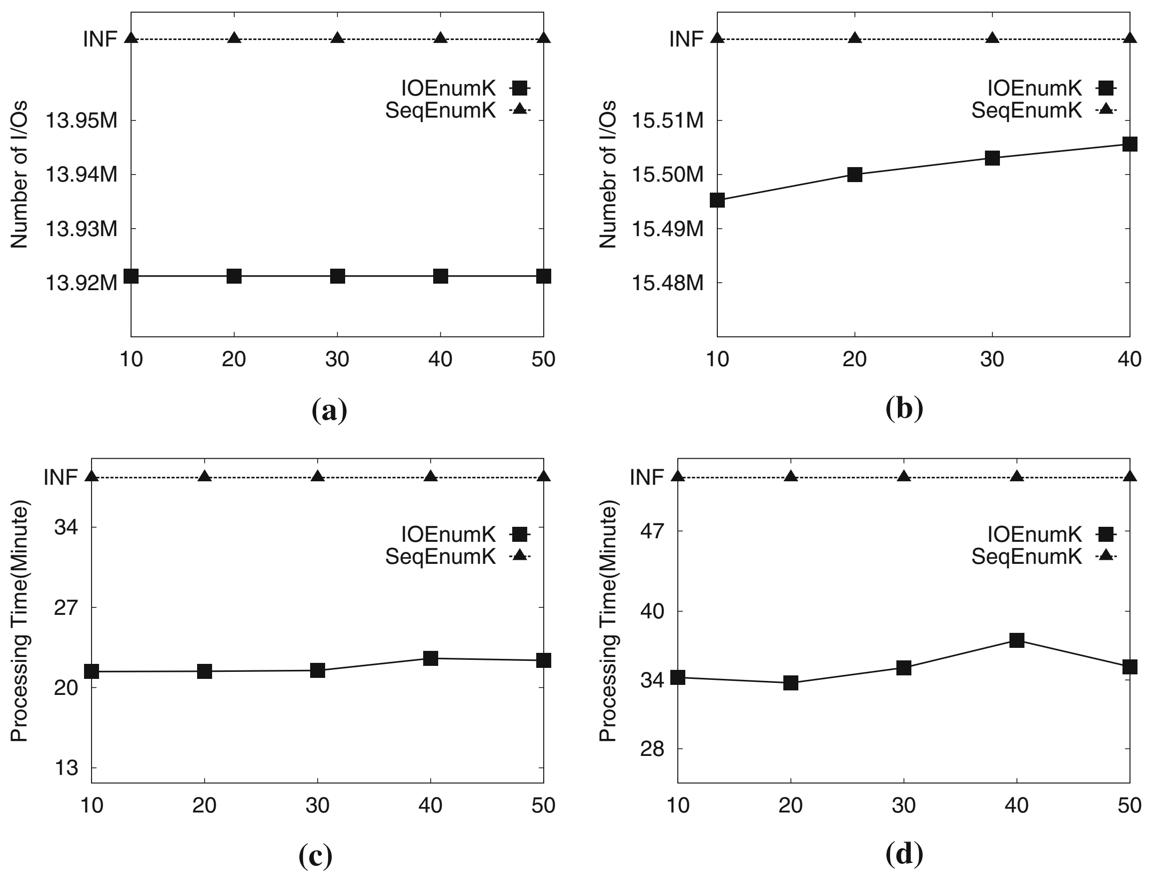

Fig. 15 Vary k (I/O efficient algorithms). **a** UK-2005 (# I/Os), **b** Webbase (# I/Os), **c** UK-2005 (time), **d** Webbase (time)

7.3 I/O efficient algorithms

Exp-10: Vary k. In this experiment, we compare the number of I/Os and the processing time for SeqEnumK and IOEnumK on two massive datasets *UK-2005* and *Webbase* when we vary k from 10 to 50. The results are shown in Fig. 15. For SeqEnumK, it cannot terminate in time limit for all k values on these two datasets. For IOEnumK, when k increases, the number of I/Os on *UK-2005* has little fluctuation (Fig. 15a), while the number of I/Os on *Webbase* increases as k increases (Fig. 15b). This is because as k increases, the pruning power of our optimization strategies decreases, then the size of G' returned by IOInitK in IOEnumK increases, thus more I/Os are needed to scan G' during the processing. Therefore the number of I/Os increases as k increases. The processing time of IOEnumK has similar trends as the number of I/Os in most cases (Fig. 15c, d). An interesting observation is that in Fig. 15d, when k increase from 40 to 50, the processing time decreases, which violates the intuition. The reason for this phenomenon is as follows: the processing time is determined by two factors: the I/Os and the number of enumerated cliques during processing. For I/Os, as explained above, the number of I/Os increases as k increases. However, for the number of enumerated cliques during processing, there is no such certain rule as I/Os. As stated in Sect. 1, the number of cliques highly depends on the input graph. Since the pruning power of our optimization strategies fluctuates as k changes, G' returned by IOInitK in IOEnumK are also quite different as k changes. Thus the numbers of enumerated cliques are quite different as k varies. In consequence, for the processing time, no certain rule exists when k changes. This explains the interesting turning point on the curve of processing time, while there is no such point on the curve of I/Os.

Exp-11: Vary α. In this experiment, we vary α from 0.1 to 0.5 and report the number of I/Os and the processing time for SeqEnumK and IOEnumK on *UK-2005* and *Webbase*. The results are shown in Fig. 16. SeqEnumK still cannot finish in the time limit on both datasets. For IOEnumK, the number of I/Os decreases when we vary α from 0.1 to 0.2 and keep stable as α increases on *UK-2005* (Fig. 16a) and the number of I/Os decreases when α increases on *Webbase* (Fig. 16b). This is because as α increases, the pruning power of our optimization strategies increases, then the size of G' return by IOInitK decreases, thus less I/Os are needed to scan G' during the processing. For the processing time, it generally decreases as α increases on both datasets (Fig. 16c, d). However, turning points still exist on the curves, such as the

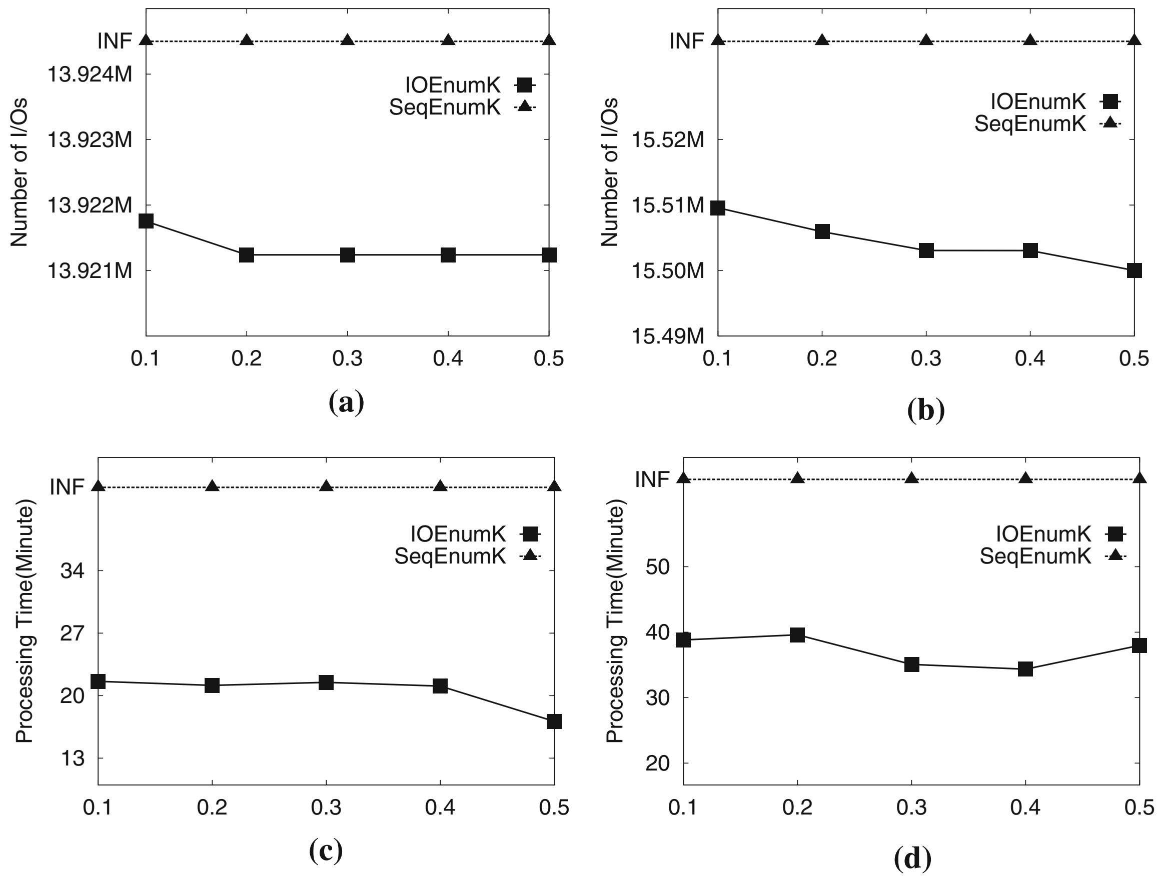

Fig. 16 Vary α (I/O efficient algorithms). **a** UK-2005 (# I/Os), **b** Webbase (# I/Os), **c** UK-2005 (time), **d** Webbase (time)

point when $\alpha = 0.4$ in Fig. 16d. The reason for this phenomenon is similar to that when we vary k in **Exp-10**. The processing time is determined by the I/Os and the number of enumerated cliques during processing. As explained above, the number of I/Os decreases when α increases. However, the number of enumerated cliques is dependent on the input graph. Since the pruning power of our optimization strategies fluctuates as α changes, G' returned by IOInitK are also quite different as k changes. Thus the numbers of enumerated cliques are quite different as α varies. In consequence, there exists no certain rules on the processing time when α changes and it is reasonable that the turning points exist on the curve of processing time.

8 Conclusion

In this paper, we study the diversified top-k clique search problem, which is to find k cliques that can cover most number of nodes in a graph. We show that it is impractical to keep all maximal cliques in memory before computing the diversified top-k cliques. Therefore, we devise a new algorithm to maintain k candidates during maximal clique enumeration. Our algorithm has limited memory footprint and can achieve a guaranteed approximation ratio. We introduce a novel PNP-Index based on which an optimal candidate maintenance algorithm is designed. We further explore three optimization strategies to avoid enumerating all maximal cliques and thus largely reduce the computational cost. Finally, we propose an I/O efficient algorithm to handle the scenario when the input graph is too large to fit into main memory. We conduct extensive performance studies on large real graphs and synthetic graphs to demonstrate the efficiency and effectiveness of our approach.

Acknowledgments Lu Qin was supported by ARC DE140100999. Xuemin Lin was supported by NSFC61232006, NSFC61021004, ARC DP120104168, and ARC DP140103578. Lijun Chang was supported by ARC DE150100563. Wenjie Zhang was supported by ARC DE120102144 and DP120104168.

References

1. Aggarwal, A., Vitter, J., et al.: The input/output complexity of sorting and related problems. Commun. ACM **31**(9), 1116–1127 (1988)

2. Agrawal, R., Gollapudi, S., Halverson, A., Ieong, S.: Diversifying search results. In: Proceedings of WSDM'09, pp. 5–14 (2009)
3. Akkoyunlu, E.A.: The enumeration of maximal cliques of large graphs. SIAM J. Comput. **2**(1), 1–6 (1973)
4. Angel, A., Koudas, N.: Efficient diversity-aware search. In: Proceedings of SIGMOD'11, pp. 781–792 (2011)
5. Ausiello, G., Boria, N., Giannakos, A., Lucarelli, G., Paschos, V.T.: Online maximum k-coverage. Discrete Appl. Math. **160**(13–14), 1901–1913 (2012)
6. Badanidiyuru, A., Mirzasoleiman, B., Karbasi, A., Krause, A.: Streaming submodular maximization: massive data summarization on the fly. In: Proceedings of KDD'14, pp. 671–680 (2014)
7. Batagelj, V., Zaversnik, M.: An o(m) algorithm for cores decomposition of networks. CoRR. cs.DS/0310049 (2003)
8. Bernard, H.R., Killworth, P.D., Sailer, L.: Informant accuracy in social network data IV: a comparison of clique-level structure in behavioral and cognitive network data. Soc. Netw. **2**(3), 191–218 (1979)
9. Berry, N., Ko, T., Moy, T., Smrcka, J., Turnley, J., Wu, B.: Emergent clique formation in terrorist recruitment. In: Workshop on Agent Organizations: Theory and Practice (2004)
10. Borodin, A., Lee, H.C., Ye, Y.: Max-sum diversification, monotone submodular functions and dynamic updates. In: Proceedings of PODS'12, pp. 155–166 (2012)
11. Bron, C., Kerbosch, J.: Finding all cliques of an undirected graph (algorithm 457). Commun. ACM **16**(9), 575–576 (1973)
12. Carraghan, R., Pardalos, P.M.: An exact algorithm for the maximum clique problem. Operat. Res. Lett. **9**(6), 375–382 (1990)
13. Chang, L., Yu, J.X., Qin, L.: Fast maximal cliques enumeration in sparse graphs. Algorithmica **66**(1), 173–186 (2013)
14. Cheng, J., Ke, Y., Chu, S., Özsu, M.T.: Efficient core decomposition in massive networks. In: Proceedings of ICDE, pp. 51–62 (2011)
15. Cheng, J., Ke, Y., Fu, A.W.-C., Yu, J.X., Zhu, L.: Finding maximal cliques in massive networks. ACM Trans. Database Syst. **36**(4), 21:1–21:34 (2011)
16. Cheng, J., Zhu, L., Ke, Y., Chu, S.: Fast algorithms for maximal clique enumeration with limited memory. In: Proceedings of KDD'12, pp. 1240–1248 (2012)
17. Chierichetti, F., Kumar, R., Tomkins, A.: Max-cover in map-reduce. In: Proceedings of WWW'10, pp. 231–240 (2010)
18. Chu, S., Cheng, J.: Triangle listing in massive networks and its applications. In: Proceedings of SIGKDD, pp. 672–680 (2011)
19. Demidova, E., Fankhauser, P., Zhou, X., Nejdl, W.: DivQ: diversification for keyword search over structured databases. In: Proceedings of SIGIR'10, pp. 331–338 (2010)
20. Deng, T., Fan, W.: On the complexity of query result diversification. ACM Trans. Database Syst. **39**(2), 15:1–15:46 (2014)
21. Drosou, M., Pitoura, E.: Search result diversification. SIGMOD Rec. **39**(1), 41–47 (2010)
22. Eppstein, D., Loffler, M., Strash, D.: Listing all maximal cliques in sparse graphs in near-optimal time. ISAAC **1**, 403–414 (2010)
23. Eppstein, D., Strash, D.: Listing all maximal cliques in large sparse real-world graphs. In: Proceedings of SEA'11, pp. 364–375 (2011)
24. Fan, W., Wang, X., Wu, Y.: Diversified top-k graph pattern matching. PVLDB **6**(13), 1510–1521 (2013)
25. Feige, U.: A threshold of ln n for approximating set cover. J. ACM **45**(4), 634–652 (1998)
26. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., San Francisco (1979)
27. Hu, X., Tao, Y., Chung, C.: I/O-efficient algorithms on triangle listing and counting. ACM Trans. Database Syst. **39**(4), 27:1–27:30 (2014)
28. Karp, R.M.: Reducibility among combinatorial problems. In: Complexity of Computer Computations. Plenum Press (1972)
29. Konc, J., Janezic, D.: An improved branch and bound algorithm for the maximum clique problem. Proteins **4**, 5 (2007)
30. Lee, C., Reid, F., McDaid, A., Hurley, N.: Detecting highly overlapping community structure by greedy clique expansion. In: Workshop on Social Network Mining and Analysis (2010)
31. Lin, X., Yuan, Y., Zhang, Q., Zhang, Y.: Selecting stars: The k most representative skyline operator. In: Proceedings of ICDE, pp. 86–95 (2007)
32. Minack, E., Siberski, W., Nejdl, W.: Incremental diversification for very large sets: a streaming-based approach. In: Proceedings of SIGIR'11, pp. 585–594 (2011)
33. Östergård, P.R.: A fast algorithm for the maximum clique problem. Discrete Appl. Math. **120**(1), 197–207 (2002)
34. Qin, L., Yu, J.X., Chang, L.: Diversifying top-k results. PVLDB **5**(11), 1124–1135 (2012)
35. Robson, J.: Finding a maximum independent set in time $O(2^{n/4})$. In: Technical report, 1251-01, LaBRI, Université de Bordeaux I (2001)
36. Saha, B., Getoor, L.: On maximum coverage in the streaming model & application to multi-topic blog-watch. In: Proceedings of SDM'09, pp. 697–708 (2009)
37. Schmidt, M.C., Samatova, N.F., Thomas, K., Park, B.-H.: A scalable, parallel algorithm for maximal clique enumeration. J. Parallel Distrib. Comput. **69**(4), 417–428 (2009)
38. Suyudi, M., Mohd, I.B., Mamat, M., Sopiyan, S., Supriatna, A.K.: Solution of maximum clique problem by using branch and bound method. Appl. Math. Sci. **8**(2), 81–90 (2014)
39. Tomita, E., Kameda, T.: An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments. J. Global Optim. **37**(1), 95–111 (2007)
40. Tomita, E., Tanaka, A., Takahashi, H.: The worst-case time complexity for generating all maximal cliques and computational experiments. Theor. Comput. Sci. **363**(1), 28–42 (2006)
41. Vieira, M.R., Razente, H.L., Barioni, M.C.N., Hadjieleftheriou, M., Srivastava, D., Traina, Jr., C., Tsotras, V.J.: On query result diversification. In: Proceedings of ICDE'11 (2011)
42. Wang, J., Cheng, J.: Truss decomposition in massive networks. PVLDB **5**(9), 812–823 (2012)
43. Wang, J., Cheng, J., Fu, A.W.-C.: Redundancy-aware maximal cliques. In: Proceedings of KDD'13, pp. 122–130 (2013)
44. Welsh, D.J.A., Powell, M.B.: An upper bound for the chromatic number of a graph and its application to timetabling problems. Comput. J. **10**(1), 85–86 (1967)
45. Xiang, J., Guo, C., Aboulnaga, A.: Scalable maximum clique computation using mapreduce. In Proceedings of ICDE'13, pp. 74–85 (2013)
46. Xu, Y., Cheng, J., Fu, A.W.-C., Bu, Y.: Distributed maximal clique computation. In: Proceedings of BigData'14, pp. 160–167 (2014)
47. Yu, H., Yuan, D.: Set coverage problems in a one-pass data stream. In: Proceedings of SDM'13, pp. 758–766 (2013)
48. Yuan, L., Qin, L., Lin, X., Chang, L., Zhang, W.: Diversified top-k clique search. In: Proceedings of ICDE'15, pp. 387–398 (2015)
49. Zhang, Z., Qin, L., Yu, J.X.: Contract & expand: I/O efficient sccs computing. In: Proceedings of ICDE, pp. 208–219 (2014)
50. Zhang, Z., Yu, J.X., Qin, L., Chang, L., Lin, X.: I/O efficient: computing sccs in massive graphs. In: Proceedings of SIGMOD, pp. 181–192 (2013)
51. Zhang, Z., Yu, J.X., Qin, L., Shang, Z.: Divide & conquer: I/O efficient depth-first search. In: Proceedings of SIGMOD, pp. 445–458 (2015)
52. Zheng, X., Liu, T., Yang, Z., Wang, J.: Large cliques in Arabidopsis gene coexpression network and motif discovery. J. Plant Physiol. **168**(6), 611–618 (2011)