Finding the Maximum Clique in Massive Graphs

Can Lu, Jeffrey Xu Yu, Hao Wei, Yikai Zhang
The Chinese University of Hong Kong, Hong Kong, China
{lucan,yu,hwei,ykzhang}@se.cuhk.edu.hk

1. ABSTRACT

Cliques refer to subgraphs in an undirected graph such that vertices in each subgraph are pairwise adjacent. The maximum clique problem, to find the clique with most vertices in a given graph, has been extensively studied. Besides its theoretical value as an NP-hard problem, the maximum clique problem is known to have direct applications in various fields, such as community search in social networks and social media, team formation in expert networks, gene expression and motif discovery in bioinformatics and anomaly detection in complex networks, revealing the structure and function of networks. However, algorithms designed for the maximum clique problem are expensive to deal with real-world networks.

In this paper, we devise a randomized algorithm for the maximum clique problem. Different from previous algorithms that search from each vertex one after another, our approach *RMC*, for the randomized maximum clique problem, employs a binary search while maintaining a lower bound $\underline{\omega_c}$ and an upper bound $\overline{\omega_c}$ of $\omega(G)$. In each iteration, *RMC* attempts to find a ω_t-clique where $\omega_t = \lfloor(\underline{\omega_c} + \overline{\omega_c})/2\rfloor$. As finding ω_t in each iteration is NP-complete, we extract a seed set S such that the problem of finding a ω_t-clique in G is equivalent to finding a ω_t-clique in S with probability guarantees ($\geq 1-n^{-c}$). We propose a novel iterative algorithm to determine the maximum clique by searching a k-clique in S starting from $k = \underline{\omega_c} + 1$ until S becomes $\emptyset$, when more iterations benefit marginally. As confirmed by the experiments, our approach is much more efficient and robust than previous solutions and can always find the exact maximum clique.

2. INTRODUCTION

Various networks ranging from social networks, collaboration networks to biological networks, have grown steadily. Cliques refer to subgraphs in an undirected graph where any two vertices in the subgraph are adjacent to each other. Since clique was introduced to model groups of individuals who know each other [31], cliques are widely used to represent dense communities in complex networks and the maximum clique problem has been extensively studied. The maximum clique problem is known to have direct applications in various fields and cliques are of vital importance in developing graph-based algorithms to analyze the structure and function of networks arising in diverse areas. We list some applications closely related to the maximum clique problem. (a) Community search in social networks and social media [36]: In social networks and social media, individuals who are familiar with each other are connected with edges, and large cliques can serve as candidates for communities, where community detection is acknowledged to be an efficient tool for analyzing the organization and function of complex networks by studying mesoscopic structures. (b) Team formation in expert networks [28]: In a social network or a collaboration network that captures the compatibility among experts as well as the individual capabilities of each expert, the problem of team formation requires to find a k-clique whose members can perfectly cover a given capability set, where k is the number of the experts. (c) Gene expression and motif discovery in bioinformatics [49, 52]: In gene co-expression networks, Co-Expression Groups (CEGs) are modeled as cliques. Motif discovery in bioinformatics and molecular biology requires to find large CEGs in gene co-expression networks. (d) Anomaly detection in complex networks [30, 6]: In such applications, cliques are used as signals of rare or anomaly events, like terrorist recruitment or web spam.

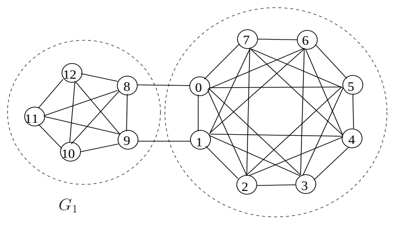


Figure 1: The maximum clique vs its relaxations

In the literature, various algorithms have been developed for the maximum clique problem, either exact or heuristic [8, 43, 44, 22, 39, 26, 21, 16, 19, 17]. However, due to the NP-hardness of the problem, all these algorithms fail when facing large and massive sparse graphs, which emerge nowadays due to the upsurge of web technologies and the Internet. In order to balance the effectiveness and efficiency, diverse relaxations of clique, for instance, quasi-clique [51], k-core [5], k-truss [46], k-edge-connectivity [2], dense subgraph [18] and many others [45, 32] have been proposed to provide good approximations of the maximum clique, which are believed explicitly or implicitly, to contain the maximum clique. However, this is not always the case.

We discuss why the relaxations cannot be effectively used to find the maximum clique always. Fig. 1 illustrates a graph G consists of two subgraphs, G_1 and G_2, along with two edges connecting them. The maximum clique of G is G_1, which is a 5-clique whereas G_2 only contains 4-cliques. Consider finding G_1 in G using some re-

Proceedings of the VLDB Endowment, Vol. 10, No. 11
Copyright 2017 VLDB Endowment 2150-8097/17/07.

laxations such as k-core, k-truss, k-edge-connected graph, and the densest subgraph. 1) G_1 is a 4-core, whereas G_2 is a 6-core. 2) G_1 is a 5-truss, whereas G_2 is a 6-truss. 3) G_1 is 4-edge-connected, whereas G_2 is 6-edge-connected. 4) G_2 is the densest subgraph of G. All these approximations will return G_2, missing the maximum clique G_1. As witnessed from Fig. 1, in applications where we need the maximum clique or large cliques, most of the relaxations may not serve the purpose. In the datasets tested (Table 1), in LiveJournal, Orkut, Foursquare, and Lastfm, the maximum truss does not contain any maximum clique entirely. Note that a graph may contain several maximum cliques.

In the literature, all the algorithms designed for discovering the maximum clique in practical networks much more rely on the implementations, such as parallel, than algorithm design [37, 40]. It is worth noticing that a shortcoming of almost all the previous algorithms for the maximum clique problem, both exact and heuristic, for both small dense artificial networks and real-life networks, is that the efficiency of the algorithms largely relies on the initial vertex ordering. In cases where the initial vertex ordering fails to extract a near-optimal maximum clique efficiently, much time will be spent on searching branches with marginal improvements.

Main contributions: We summarize the main contributions of our work as follows. First, due to the NP-hardness of the maximum clique problem, diverse relaxations of clique, for instance, k-core, k-truss, dense subgraph and many others, are proposed to provide good approximations of the maximum clique. It is interesting to find that there are some exceptions that the maximum truss does not contain any maximum clique entirely. Second, unlike previous exact algorithms that employ the branch-and-bound schema, i.e., branch from each vertex to enumerate all maximal cliques and prune fruitless branches, we propose a new binary search schema. Specifically, our approach maintains a lower bound $\underline{\omega_c}$ and an upper bound $\overline{\omega_c}$ of $\omega(G)$ and attempt to find a ω_t-clique in each iteration, where $\omega_t = \lfloor(\underline{\omega_c} + \overline{\omega_c})/2\rfloor$, delaying the brute-force search to the moment any more iterations barely work. Third, since each iteration in the binary search is actually a k-clique problem, which is NP-complete, we utilize uniform sampling to extracts a seed set S s.t., finding a ω_t-clique in G is equivalent to finding a ω_t-clique in S with probability guarantees ($\geq 1 - n^{-c}$). For each seed in S, we introduce algorithms *scSeed* and *tciSeed* to iteratively shrink its subgraphs. Fourth, we propose a new iterative brute-force algorithm *divSeed* to determine the maximum clique after the binary search. The algorithm *divSeed* attempts to find a k-clique in each seed in S iteratively, starting from $k = \underline{\omega_c} + 1$. Such constrains provide great power for pruning. Fifth, we conduct extensive experimental studies to show the robustness and efficiency of our approach.

Organization: The preliminaries and the problem statement are given in Section 3. We discuss related works in Section 4, and review the previous algorithms in Section 5. We give an overview of our approach in Section 6, and discuss the algorithms in Section 7. We have conducted comprehensive experimental studies and report our findings in Section 8. We conclude this paper in Section 9.

3. PROBLEM DEFINITION

In this paper, a social network is modeled as an undirected graph $G = (V, E)$ without self-loops or multiple edges, where V and E denote the sets of vertices and edges of G, respectively. We use n and m to denote the numbers of vertices and edges of G, respectively, i.e., $n = |V|$ and $m = |E|$. In the paper, we assume without loss of generality that G is connected, otherwise the algorithm can be applied to each connected component in a graph. For a vertex $u \in V$, the neighbors of u are denoted as $\Gamma(u)$ such that $\Gamma(u) = \{v \mid (u, v) \in E\}$, and the degree of u is denoted as $\delta(u) = |\Gamma(u)|$. Similarly, for a set of vertices $S \subseteq V$, $\Gamma(S) = \{u \in V \setminus S \mid \exists v \in S, (u, v) \in E\}$. We also define $\Lambda(S)$ to be the set of common neighbors of vertices in S, i.e., $\Lambda(S) = \{u \in V \setminus S \mid \forall v \in S, (u, v) \in E\}$.

A graph G is a clique if there are edges between any two vertices in G. We also call a vertex set $C \subseteq V$ a clique if the subgraph induced by C is a clique. C is a maximal clique if there exists no proper superset of C that is also a clique and C is a maximum clique if there exists no clique C' such that $|C'| > |C|$. The number of vertices in a maximum clique in graph $G = (V, E)$ is denoted as $\omega(G)$ or $\omega(V)$. For simplicity, in the following discussion, we use $\overline{\omega}(G)$ or $\overline{\omega}(V)$ to denote the upper bound of the maximum clique of $G = (V, E)$ and use $\underline{\omega}(G)$ or $\underline{\omega}(V)$ to represent the lower bound of the maximum clique of G, respectively.

Upper bound: Vertex coloring is widely used to obtain an upper bound of $\omega(G)$. A vertex coloring is to assign a color to every vertex in graph G, and a coloring is a proper coloring if any two adjacent vertices are assigned with different colors. A minimum coloring is a proper coloring that uses the least number of colors. Let the size of the minimum coloring of $G = (V, E)$ be $\chi(G)$ or $\chi(V)$. We have $\chi(G) \geq \omega(G)$ since a proper coloring of the maximum clique K needs $\omega(G)$ colors and $\chi(V) \geq \chi(S)$ for $S \subset V$. For simplicity, in the following discussion, a vertex coloring always refers to a proper coloring. It is known that k-core and k-truss are also widely used as upper bounds of $\omega(G)$. Specifically, if the maximum core of G is a k-core, then $\omega(G) \leq k + 1$. Similarly, if the maximum truss of G is a k-truss, then $\omega(G) \leq k$.

Lower bound: An independent set of graph $G = (V, E)$ is a subset $W \subset V$ such that $(u, v) \notin E$ for any two vertices $u, v \in W$. A maximum independent set is an independent set of the largest possible size denoted as $o(G)$. A lower bound of $\omega(G)$ can be given as $\omega(G) = o(\overline{G})$, where $\overline{G} = (V, E')$ is the complementary graph of G, i.e., for any non self-loop edge (u, v), $(u, v) \in E'$, if $(u, v) \notin E$.

Randomized Maximum Clique Problem. In this paper, we study Randomized Maximum Clique Problem, i.e., given a graph G, find a clique S of size $\omega(G)$ with high probability $\geq 1 - n^c$ where c is a constant.

Problem Hardness. The clique decision problem, i.e., to return a Boolean value indicating whether graph G contains a k-clique, is one of the Karp's 21 NP-complete problems [23], and the maximum clique problem is both fixed-parameter intractable and hard to approximate [20], which implies that unless P=NP, there cannot be any polynomial time algorithm that approximates the maximum clique within a factor better than $O(n^{1-\epsilon})$, for any $\epsilon > 0$.

4. RELATED WORKS

The "clique" was first studied to model groups of individuals who know each other [31], and the maximum clique problem has been extensively studied by researchers. Cook [11] and Karp [23] utilize the theory of NP-completeness and related intractability results to provide a mathematical explanation for the difficulty of the maximum clique problem. Tarjan and Trojanowski [42] investigate the problem from the viewpoint of worst-case analysis. In the 1990s, Feige et al. [15] prove that it is impossible to approximate the problem accurately and efficiently. Later, Feige [14] proposes a polynomial-time algorithm that finds a clique of size $O((\frac{logn}{loglogn})^2)$ whenever the graph has a clique of size $O(\frac{n}{\log n^b})$ for any constant b. Analogously, Håstad shows that, unless P=NP, there cannot be any polynomial time algorithm that approximates the maximum clique within a factor better than $\emptyset(n^{1-\epsilon})$, for any $\epsilon > 0$.

Most of the exact maximum clique searching algorithms are based on branch and bound search. One of the first and most classical algorithms, proposed by Carraghan and Pardalos [8], prunes branches whose further expansion fails to find a larger clique than the current optimal one. Another effective algorithm named MCR, proposed by Tomita and Kameda [43], uses graph coloring to obtain an upper bound on the size of the maximum clique for each branch. In the MCS algorithm [44], a novel routine that tries to recolor vertices with the biggest color into a smaller one is introduced to improve the pruning performance. An introduction and computational study can be found in [38].

While the majority of the algorithms for the maximum clique problem, including the above mentioned ones, concentrate on small dense graphs, only a few works, e.g. [37, 40], study it for large and massive sparse graphs. Both [37] and [40] utilize an adjacency list representation of graphs and unroll the first level of the search tree to enforce the early pruning. The main difference between these two works is that [37] uses bounds based on degrees while [40] employs tighter bounds with the concept of k-core [41]. To further improve the performance, both [37] and [40] employ a bit-parallel algorithm that uses bitstrings to encode sparse adjacency list.

Since the maximum clique problem is NP-hard, much effort has recently been devoted to developing efficient heuristics, that are meaningful in practice but lack performance guarantees. Sequential greedy heuristics either try to generate a clique by iteratively adding a vertex to a current clique or to find a clique by repeatedly removing a vertex from the current vertex set which is not a clique [26, 22]. Since sequential greedy heuristics can find only one maximal clique, local search heuristics are proposed to improve approximation solutions by, for instance, a (j, k)-swap, which replaces j vertices from the current maximal clique by other k vertices. Pullan and Hoos [39] introduce dynamic local search which combines fast neighborhood search and usage of penalties to promote diversification. Other heuristics include simulated annealing [21], GEASP [16], tabu search [19] and the neural networks [17].

Another similar topic related to maximum clique problem is maximal clique enumeration. The maximum clique can be obtained through scanning all maximal cliques for the largest one. The first algorithms for the maximal clique enumeration problem are the backtracking method [3, 7]. To further reduce the search space, [24] employs effective pruning strategy by selecting good pivots. As the massive size of various graphs has outpaced the advance in the memory available on commodity hardware, Cheng et al. propose efficient algorithms [9, 10] to reduce both the I/O cost and CPU cost of maximal clique enumeration in massive networks. To deal with the excessive size and overlapping parts in classic maximal clique enumeration, Wang et al. introduce the notion of τ-visible MCE [47] to reduce the redundancy while capturing the major information in the result, and Yuan et al. study the diversified top-k clique search problem [50] which is to find top-k cliques that can cover most number of nodes in the graph.

5. THE PREVIOUS ALGORITHMS

The main idea underlying all previous algorithms is given as follows. Let the maximum clique found, denoted as C_m, as a lower bound of $\omega(G)$. Let C denote the current clique, and $P = \Lambda(C)$ denote the candidate set from which a vertex will be selected for C to grow next. Suppose there is a function $U(C, P)$ which returns an upper bound of $\omega(C \cup P)$, i.e., the size of the maximum clique that can be found by selecting any additional vertex in P to grow from C. If $U(C, P) > |C_m|$, it continues to grow C by selecting one vertex from P. Otherwise, it stops searching from the current C. In other words, it prunes branches that cannot generate cliques larger than C_m. The two main issues lying on the development of the previous algorithms are: finding a near-maximum clique as fast as possible and designing a high quality function $U(C, P)$.

Algorithm 1 *MaxClique* (G)

```
1: $C_m \leftarrow \emptyset$, sort $V$ in some specific ordering $o$;
2: for $i = 1$ to $n$ according to $o$ do
3:     $C \leftarrow \{v_i\}$, $P \leftarrow \{v_{i+1}, \ldots, v_n\} \cap \Gamma(v_i)$;
4:     sort $P$ in some ordering $o'$;
5:     for $v_j \in P$ according to the sorting order $o'$ do Clique($G, C, P, v_j$);
6: end for
7: return $C_m$;

8: Procedure Clique($G, C, P, v_j$)
9: $C \leftarrow C \cup \{v_j\}$, $P \leftarrow P \cap \Gamma(v_j)$;
10: if $P = \emptyset$ then
11:     if $|C| > |C_m|$ then $C_m \leftarrow C$;
12: else if $U(C, P) > |C_m|$ then
13:     sort $P$ in some ordering $o''$;
14:     for $v_k \in P$ according to the sorting order $o''$ do Clique($G, C, P, v_k$);
15: end if
```

Algorithm 1 illustrates the schema of previous algorithms. First, it initializes C_m as $\emptyset$ and sorts vertices in V in some specific ordering (Line 1). With such ordering, booktracking search from v_i considers only vertex set $\{v_i, v_{i+1}, \ldots, v_n\}$, i.e., initialize $C \leftarrow \{v_i\}$ and $P \leftarrow \{v_{i+1}, \ldots, v_n\} \cap \Gamma(v_i)$ (Line 3). Similarly, vertices in P are also sorted in some ordering (Line 4). Then, branching from v_i iteratively adds a candidate vertex v_j from P to C by invoking the procedure Clique(G, C, P, v_j), updating C_m or pruning fruitless branches (Line 5). After processing all vertices, the resulting C_m by processing all vertices in V is the maximum clique of G (Line 7). Here, the procedure Clique(G, C, P, v_j) enumerates all maximal cliques branching from v_j, and attempts to either improve C_m or prune fruitless branches. We explain the procedure Clique below. First, it updates C and P by adding v_j to C and condensing P through setting $P \leftarrow P \cap \Gamma(v_j)$ (Line 9). Second, if $P = \emptyset$, a maximal clique is found, and C_m will be updated if $|C| > |C_m|$ (Line 10-11). Third, it compares the lower bound $|C_m|$ with the upper bound $U(C, P)$. If $U(C, P) > |C_m|$, it branches from vertices in P recursively (Line12-14). Otherwise, it implies that it cannot find a clique larger than C_m by growing a vertex in P from C.

There are two issues in the schema as shown in Algorithm 1: the function $U(C, P)$ and the vertex sorting (Line 1, Line 4, and Line 13). We discuss them below. First, as an upper bound of the branch, $U(C, P)$ answers whether the combination of C and P has a chance to unseat the maximum clique C_m found so far. A tight bound $U(C, P)$ benefits pruning fruitless branches as fast as possible. The simplest upper bound $|C| + |P|$ is used by [8, 13] such that the branch can be abandoned if $|C| + |P| \leq |C_m|$. Later, [43] proposes graph coloring on P. Let the color of any vertex $v \in P$ be $c(v)$ and the total colors used in P be $c(P)$. Such a method has two significant advances. On one hand, the largest possible clique in $C \cup P$ is a $(|C| + c(P))$-clique. On the other hand, it expands from $\{v\}$ in a branch by including only vertices colored $< c(v)$, so that the upper bound of the cliques expanding from $C \cup \{v\}$ is bounded by $|C| + c(v)$. [44] introduces a recolor mechanism. It attempts to reduce the colors used by exchanging different color classes between vertices, when coloring vertices. Recently, degrees and cores are utilized to get tighter bounds [37, 40]. Second, vertices ordering plays a significant role in generating a near-maximum clique at the very beginning. In [8], vertices are sorted in a non-decreasing degree order. By reversing the order in which search is done by [8], [35] improves the performance on random and DIMACS benchmark graphs. In order to deal with ties, [43] suggests a non-decreasing degree with tie-breaking on the sum of the degrees of adjacent vertices. In addition, [43] further suggests to process vertices in candidate set P in decreasing color order.

The previous algorithms perform well on many networks. However, they are encountering greater and greater challenges when dealing with rapidly growing networks in real life. Deep inside the existing solutions and real-world massive graphs, we observe three main reasons that harm the efficiency and robustness of the existing algorithms tremendously. First, the candidate set P is large and sparse in massive graphs. $U(C, P)$ returns a bound which is far from tight. Second, in practical massive networks, high degree vertices always have a large number of edges connecting with low degree vertices. It undermines the effectiveness and robustness of sorting strategies based on degrees significantly. Third, no upper bound of $\omega(G)$ is used in the existing algorithms. It wastes time to search remaining branches, even when the maximum clique is found at the very beginning.

6. AN OVERVIEW OF OUR APPROACH

In order to solve the shortcomings of the previous algorithms, in this paper, from a totally different viewpoint, we devise a novel randomized algorithm based on binary search, to improve the robustness and efficiency of the algorithm significantly. Similar to the conventional binary search, our algorithm maintains a lower bound $\underline{\omega_c}$ and an upper bound $\overline{\omega_c}$ of $\omega(G)$, and attempts to find a ω_t-clique in each iteration, where $\omega_t = \lfloor(\underline{\omega_c} + \overline{\omega_c})/2\rfloor$. However, there are some vital differences between our binary search and the conventional binary search. In the conventional binary search, in an iteration, it finds a value in the middle between the min and the max where data is sorted, and decides how to find the next in a half-interval. In our problem setting, in an iteration, it is to find a ω_t-clique and then locates the searching interval of $\omega(G)$ in $[\omega_t + 1, \overline{\omega_c}]$ or $[\underline{\omega_c}, \omega_t - 1]$ for the next iteration. However, finding a ω_t-clique is a k-clique problem, which is NP-complete itself. In our binary search, instead of finding a ω_t-clique in an iteration directly, we find a set of subgraphs where ω_t-cliques can exist. The set is denoted as follows.

$$S = \{(s_1, \Lambda(s_1)), (s_2, \Lambda(s_2)), \ldots, (s_i, \Lambda(s_i)), \ldots\}$$

where $(s_i, \Lambda(s_i))$ is a subgraph consisting of a clique s_i and a candidate set $\Lambda(s_i)$ from which the clique can grow. For simplicity, we use the term "seed" to represent both the clique s_i and the subgraph $(s_i, \Lambda(s_i))$. In addition, each $(s_i, \Lambda(s_i))$ is associated with a lower bound $\underline{\omega_i}$ and an upper bound $\overline{\omega}_i$ of $\omega(s_i \cup \Lambda(s_i))$, and all such bounds can be combined to update $\underline{\omega_c}$ and $\overline{\omega_c}$, s.t., $\underline{\omega_c} = max_i\{\underline{\omega_i}\}$ and $\overline{\omega_c} = max_i\{\overline{\omega}_i\}$, which in return prunes fruitless seeds.

We determine the lower and upper bounds for the next iteration. Let $\underline{\omega_c}$ and $\overline{\omega_c}$ be the current bounds, and let $\underline{\omega_p}$ and $\overline{\omega_p}$ be the previous bounds. There are 4 cases as illustrated in Fig. 2 based on S, $\underline{\omega_c}$ and $\overline{\omega_c}$. 1) The optimal case: if $S = \emptyset$ and $\underline{\omega_c} \geq \omega_t$, which is equivalent to $\underline{\omega_c} = \overline{\omega_c}$, the maximum clique has been found, the algorithm terminates. 2) If $S = \emptyset$ and $\underline{\omega_c} < \omega_t$, then G contains no ω_t-cliques, decrease $\overline{\omega_c}$ as $\omega_t - 1$. 3) If $S \neq \emptyset$, $\underline{\omega_c} \neq \underline{\omega_p}$ or $\overline{\omega_c} \neq \overline{\omega_p}$, more iterations may improve the bounds. 4) If $S \neq \emptyset$, $\underline{\omega_c} = \underline{\omega_p}$ and $\overline{\omega_c} = \overline{\omega_p}$, further iterations introduce marginal improvements, it terminates the binary search and applies a brute-force search algorithm.

We discuss how to find a ω_t-clique in G using a seed set $S = \{(s_1, \Lambda(s_1)), (s_2, \Lambda(s_2)), \ldots, (s_i, \Lambda(s_i)), \ldots\}$. Recall that s_i is a clique, $\Lambda(s_i)$ is a set of candidate from which a vertex is selected to grow the clique represented by s_i, and $(s_i, \Lambda(s_i))$ is a subgraph containing the clique s_i and $\Lambda(s_i)$.

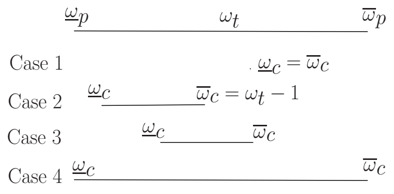


Figure 2: The interval of $\omega(G)$ in next iteration

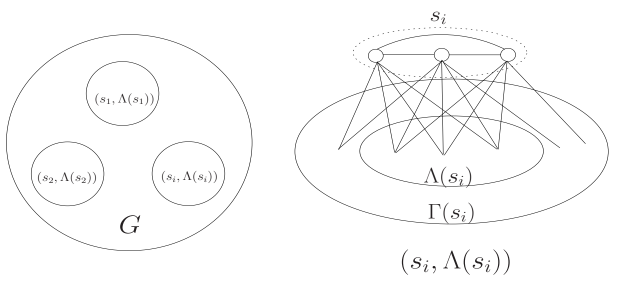


Figure 3: Find a ω_t-clique from a seed set S instead of G

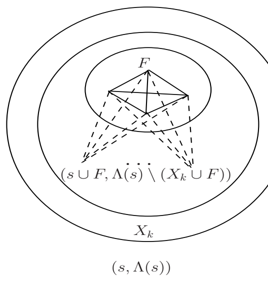
First, we conduct uniform sampling on the edges in G with probability p. Let E_s be a set of edges sampled from E. A seed s_i is an open triangle such that $s_i = (u, v, w)$ where (u, v) and (u, w) are in E_s, and (v, w) is in E. As proved by Theorem 7.1, with some specific sampling probability p, each ω_t-clique contains at least one seed with high probability. Fig. 3 illustrates the idea of finding a ω_t-clique in seed set S with probability guarantees, instead of finding a ω_t-clique in G directly. On the left, there is a set of seeds, S, sampled in G. On the right, it shows a subgraph of a seed $(s_i, \Lambda(s_i))$ in S. Here, s_i represents three vertices in a triangle, and $\Lambda(s_i)$ is a set of vertices in which every vertex is connected to each of the three vertices in s_i. Note $\Lambda(s_i) \subseteq \Gamma(s_i)$. It is worth noting that seeds in S can overlap. We explain why uniform sampling works well. a) In cases where $\omega_t \gg \omega(G)$ or G is loosely connected, uniform sampling extracts only a few open triangles. As a consequence, we can determine $S = \emptyset$ quickly, implying $\omega(G) < \omega_t$. b) Links in real-world networks exhibit inhomogeneities and community structure. With such characteristics, uniform sampling can accurately partition the whole graph into several dense clusters. As bounds $\underline{\omega_i}$ and $\overline{\omega}_i$ provide accurate information for each seed $(s_i, \Lambda(s_i))$, sorting based on such bounds is more accurate and robust than sorting based on degrees.

Second, for each seed $(s, \Lambda(s)) \in S$, we iteratively filter vertices in $\Lambda(s)$ that cannot appear in a potential ω_t-clique with s, and move some vertices from $\Lambda(s)$ to s to construct a potential maximum clique in $(s \cup \Lambda(s))$. In other words, for each seed $(s, \Lambda(s)) \in S$, we enlarge s and shrink $\Lambda(s)$ until $\Lambda(s) = \emptyset$ to find a potential maximum clique in three main approaches, repeatedly.

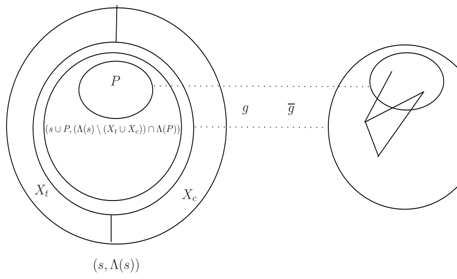
The reduction by k-core: It is based on k-core to enlarge s and shrink $\Lambda(s)$ as follows.

$$(s, \Lambda(s_i)) \rightarrow (s \cup F, \Lambda(s) \setminus (X_k \cup F)) \quad (1)$$

Here X_k is a subset of $\Lambda(s)$ that cannot exist in a $(k - |s| - 1)$-core in $\Lambda(s)$ for a given k, and F is a set of vertices that have a potential to exist in a maximum clique of $\Lambda(s)$ where $F \subseteq \Lambda(s) \setminus X_k$. There are two conditions for F to be selected. The first condition is that every vertex in F is connected to every vertex in s_i, which is ensured by the definition of $\Lambda(s_i)$. The second condition is that every vertex in F is connected to every other vertex in $\Lambda(s) \setminus X_k$. In other words, F is a clique to be considered. We illustrate it

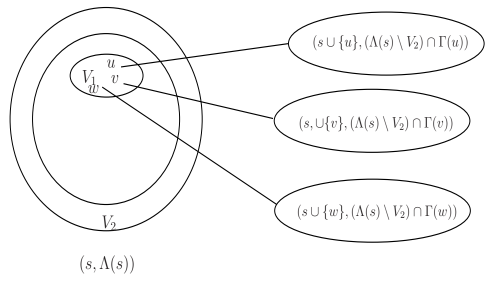


(a) The 1st by k-core



P
$(s \cup P, (\Lambda(s) \setminus (X_t \cup X_c)) \cap \Lambda(P))$
g
$\overline{g}$
X_t
X_c
$(s, \Lambda(s))$

(b) The 2nd by k-truss, coloring, and independent set



u
V_1
v
w
V_2
$(s, \Lambda(s))$
$(s \cup \{u\}, (\Lambda(s) \setminus V_2) \cap \Gamma(u))$
$(s, \cup \{v\}, (\Lambda(s) \setminus V_2) \cap \Gamma(v))$
$(s \cup \{w\}, (\Lambda(s) \setminus V_2) \cap \Gamma(w))$

(c) The 3rd by dividing seeds

Figure 4: The Three Reductions

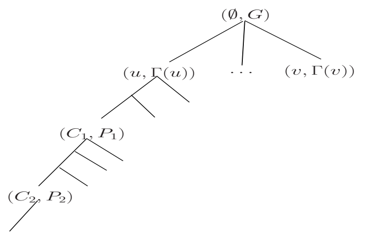
in Fig. 4(a). The first approach is done in the *scSeed* algorithm (Algorithm 4) together with the sampling.

The reduction by k-truss, coloring, and independent set: It is based on k-truss, coloring, and independent set to enlarge s and shrink $\Lambda(s)$ as follows, given $(s, \Lambda(s))$ has been enlarged/shrunk by the first approach.

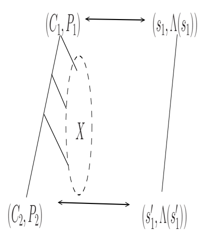
$$(s, \Lambda(s)) \rightarrow (s \cup P, (\Lambda(s) \setminus (X_t \cup X_c)) \cap \Lambda(P)) \qquad (2)$$

By the k-truss, we identify X_t which is a subset of $\Lambda(s)$ that cannot exist in a $(\omega_t - |s|)$-truss in $\Lambda(s)$. By coloring, we identify $X_c \subseteq \Lambda(s) \setminus X_t$ that contains vertices whose neighbors can be colored with less than $\omega_t - |s| - 1$ colors. Hence, the clique to be found does not contain vertices in $X_t \cup X_c$. Fig. 4(b) illustrates the main idea. Let the largest circle on the left represent $(s, \Lambda(s))$. The second largest circle on the left represents a subgraph g of $(s, \Lambda(s))$ by excluding the vertices in $(X_t \cup X_c)$. We further reduce g using independent set. Let $\overline{g}$ be the complementary graph of g (the largest circle on the right). With the help of independent set found among $\overline{\Lambda}(s) \setminus (X_t \cup X_c)$ in $\overline{g}$, we can extract a subset $P \subseteq \Lambda(s) \setminus (X_t \cup X_c)$ in g. As proved by Theorem 7.3, P belongs to one of the maximum independent sets of $\overline{g}$. Therefore, P belongs to one of the maximum cliques of g, i.e., $\Lambda(s) \setminus (X_t \cup X_c)$. As a result, we further enlarge s by P, and shrink $\Lambda(s)$ accordingly. The second approach is done in the *tciSeed* algorithm (Algorithm 5). It is worth noting that the reduction by k-truss is more powerful than the reduction by k-core, since k-trusses are contained in $(k-1)$-cores, whereas truss decomposition costs $O(m^{1.5})$, more expensive than core decomposition, which costs $O(m)$.

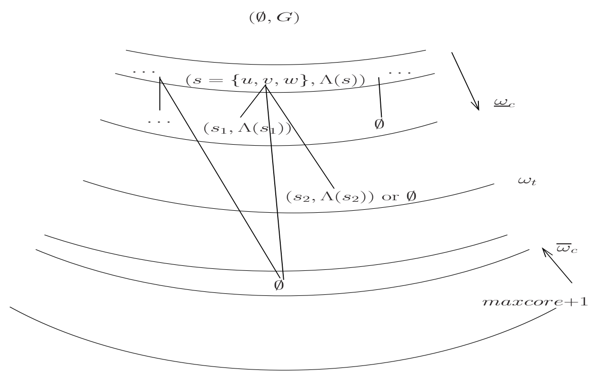
The reduction by dividing: Let $(s, \Lambda(s))$ be a subgraph where s cannot be enlarged by the first and the second approaches. We propose a new optimized brute-force search algorithm. We iteratively find k-cliques in each seed $(s, \Lambda(s)) \in S$ starting from $k = \underline{\omega_c} + 1$, and we find the maximum clique when S becomes $\emptyset$. To process a seed $(s, \Lambda(s))$, with the help of k, we extract two subsets $V_1, V_2 \subseteq \Lambda(s)$ by graph coloring. Here, V_2 represents the vertices whose neighbors can be colored with $< k - |s| - 1$ colors. V_1 is a subset of $\Lambda(s) \setminus V_2$ representing the vertices that are colored $\geq k - |s|$. As a result, the vertices in V_1 can possibly be in the maximum clique, but the vertices in V_2 cannot. Instead of finding the maximum clique in a seed $(s, \Lambda(s))$, we find the maximum clique using a series of smaller/denser seeds $(s \cup \{u\}, (\Lambda(s) \setminus V_2) \cap \Gamma(u))$ for every $u \in V_1$. Fig. 4(c) illustrates the idea. In Fig. 4(c), the vertices V_2 will be excluded. Suppose there are 3 vertices, u, v, and w, in V_1. We further compute the maximum clique in the 3 corresponding small/denser seeds. The third approach is done in the *divSeed* algorithm (Algorithm 7).



(a) The *DFS* search tree T_d



(b) T_d vs T_b



(c) T_b

Figure 5: The *BFS* search tree T_b

Search Trees: We discuss the main differences between the existing branch-and-bound approaches and our new approach in terms of search trees, in finding the maximum cliques in massive graphs. Let T_d and T_b denote the search tree by the existing branch-and-bound approaches and ours, respectively. We use node instead of vertex when we discuss trees. In T_d, a node represents a pair (C, P), where C is a growing clique and P is its candidate set. In T_b, a node represents a seed $(s, \Lambda(s))$, where the root of both trees is $(\emptyset, G)$. A node in both trees represents the same information, because s is a clique and $\Lambda(s)$ is its candidate set. Let a child node of (C, P) be (C', P') in T_d, and let a child node of $(s, \Lambda(s))$ be $(s', \Lambda(s'))$ in T_b. In T_d, $C' = C \cup \{u\}$ where u is selected from P, and $P' = \Lambda(C) \cap \Gamma(u)$. The existing branch-and-bound approaches conduct *DFS* over T_d (Fig. 5(a)). In T_b, $s' = s \cup s_0$ and $\Lambda(s') = \Lambda(s) \cap \Lambda(s_0)$, where $s_0 \subset \Lambda(s)$. We conduct *BFS* over T_b. We show the differences between T_d and T_b in Fig. 5(b). It is worth noting that in T_b it attempts to select more vertices from $\Lambda(s)$ to enlarge a growing clique s in a node when it branches to its child node, whereas in T_d it does it by selecting a single vertex v from P. In other words, an edge in T_b represents a path in T_d. In addition, as shown in Fig. 5(b), the vertices in X in T_d can be pruned by our approach in T_b in an early stage. All is because un-

Algorithm 2 *RMC* (G)

```
1: $(r, \underline{\omega_c}, \overline{\omega_c}, C_m) \leftarrow Init(G)$;
2: if $r = 1$ then return $C_m$;
3: while $\overline{\omega_c} - \underline{\omega_c} \geq \theta$ do
4:     $\omega_t \leftarrow \lfloor(\overline{\omega_c} + \underline{\omega_c})/2\rfloor$, $\overline{\omega_p} \leftarrow \overline{\omega_c}$, $\underline{\omega_p} \leftarrow \underline{\omega_c}$;
5:     $(\underline{\omega_c}, \overline{\omega_c}, C_m, S) \leftarrow scSeed(G, \omega_t)$;
6:     switch (states of $S, \underline{\omega_c}, \omega_t$)
7:     case $S = \emptyset$ and $\underline{\omega_c} \geq \omega_t$: return $C_m$;
8:     case $S = \emptyset$ and $\underline{\omega_c} < \omega_t$: $\overline{\omega_c} \leftarrow \omega_t - 1$; break;
9:     case $S \neq \emptyset$:
10:        $(\underline{\omega_c}, \overline{\omega_c}, C_m, S) \leftarrow tciSeed(G, S, \omega_t)$;
11:        switch (states of $S, \underline{\omega_c}, \omega_t, \underline{\omega_p}, \overline{\omega_p}$)
12:        case $S = \emptyset$ and $\underline{\omega_c} \geq \omega_t$: return $C_m$;
13:        case $S = \emptyset$ and $\underline{\omega_c} < \omega_t$: $\overline{\omega_c} \leftarrow \omega_t - 1$; break;
14:        case $\underline{\omega_c} = \underline{\omega_p}$ and $\overline{\omega_c} = \overline{\omega_p}$: terminate the while loop;
15:        end switch
16:    end switch
17: end while
18: $\omega_t \leftarrow \underline{\omega_c} + 1$;
19: $(\underline{\omega_c}, \overline{\omega_c}, C_m, S) \leftarrow scSeed(G, \omega_t)$;
20: if $S = \emptyset$ then return $C_m$;
21: $(\underline{\omega_c}, \overline{\omega_c}, C_m, S) \leftarrow tciSeed(G, S, \underline{\omega_c} + 1)$;
22: while $S \neq \emptyset$ do $(\underline{\omega_c}, \overline{\omega_c}, C_m, S) \leftarrow divSeed(G, S, \underline{\omega_c} + 1)$;
23: return $C_m$;
```

promising candidates in $\Lambda(s)$ will be pruned as soon as possible. Next, we discuss the binary search by T_b. As shown in Fig. 5(c), initially, the s of any $(s, \Lambda(s))$ in the initial seed set S is an open triangle based on which the lower bound $\underline{\omega_c}$ is given, and the upper bound $\overline{\omega_c}$ is $maxcore + 1$, and $\omega_t = \lfloor(\overline{\omega_c} + \underline{\omega_c})/2\rfloor$. In every iteration in the binary search, it updates $\underline{\omega_c}$ and $\overline{\omega_c}$ by exploring the current seed set S. First, $\underline{\omega_c}$ moves downwards where at least one $\underline{\omega_c}$-clique exists, which implies $\omega(G) \geq \underline{\omega_c}$. Second, $\overline{\omega_c}$ moves upwards where no $(\overline{\omega_c} + 1)$-cliques, which implies $\omega(G) \leq \overline{\omega_c}$. Third, ω_t indicates that it is possible to find ω_t-cliques between $\underline{\omega_c}$ and $\overline{\omega_c}$. By finding a ω_t-clique, we can move $\underline{\omega_c}$ to ω_t, and by finding all empty seeds in ω_t, we can move $\overline{\omega_c}$ to $\omega_t - 1$. Note that there is no need to construct the whole T_b tree.

7. THE NEW APPROACH RMC

We give our *RMC* ((Randomized Maximum Clique) algorithm in Algorithm 2. In Algorithm 2, the algorithm *Init* (Algorithm 3) initializes $\underline{\omega_c}$, $\overline{\omega_c}$, and C_m, and returns a variable r indicates whether the maximum clique is already found in the graph G (Line 1). If $r = 1$, then C_m is the maximum clique, the algorithm terminates by returning C_m (Line 2). Otherwise, it applies a binary search to reduce the gap between the lower bound $\underline{\omega_c}$ and upper bound $\overline{\omega_c}$ in a while loop (Line 3-17). The loop will continue if the difference between $\underline{\omega_c}$ and $\overline{\omega_c}$ is greater than or equal to a threshold θ. In other words, the binary search will stop, if further iterations will not reduce the search space significantly. In each iteration, we attempt to find a ω_t-clique, where $\lfloor(\overline{\omega_c} + \underline{\omega_c})/2\rfloor$ (Line 4). First, we use a randomized algorithm *scSeed* (Algorithm 4) to update $\underline{\omega_c}$ and $\overline{\omega_c}$, and obtain a seed set S where ω_t-cliques possibly exist (Line 5). Here, *scSeed* applies uniform sampling on the edges of G to extract seeds $s_i = (u, v, w)$ s.t. (u, v) and (u, w) are sampled and (v, w) is an edge in G. It is worth noting that a seed is an open triangle. As proved by Theorem 7.1, with some specific sampling probability p, each ω_t-clique contains at least one seed with high probability. For each seed s_i, *scSeed* applies the core decomposition [5] on $\Lambda(s_i)$, prunes vertices that exist in no ω_t-cliques, and moves vertices that connected to all others to s_i. This results in an upper bound $\overline{\omega}_i$ and a lower bound $\underline{\omega}_i$ of $\omega(s_i \cup \Lambda(s_i))$. We use such bounds obtained for seeds together to update $\underline{\omega_c}$ and $\overline{\omega_c}$,

Algorithm 3 *Init* (G)

```
1: compute core$(u)$ for every $u$ in $G$;
2: let the max core number be $cm$, $\overline{\omega_c} \leftarrow cm + 1$;
3: if the max core of $G$ is a clique then
4:     $C_m \leftarrow$ max core of $G$, $\underline{\omega_c} \leftarrow |C_m|$;
5:     return $(1, \underline{\omega_c}, \overline{\omega_c}, C_m)$;
6: end if;
7: for each vertex $v$ with core$(v) \geq \underline{\omega_c}$ in non-ascending order of core numbers do
8:     greedily generate a maximal clique $C$ starting from $v$;
9:     $C_m \leftarrow C$, $\underline{\omega_c} \leftarrow |C_m|$ if $|C| > \underline{\omega_c}$;
10: end for
11: if $\underline{\omega_c} = \overline{\omega_c}$ then return $(1, \underline{\omega_c}, \overline{\omega_c}, C_m)$;
12: compute graph coloring, and the color number as $cn$;
13: if $\overline{\omega_c} > cn$ then $\overline{\omega_c} \leftarrow cn$;
14: if $\underline{\omega_c} = \overline{\omega_c}$ then return $(1, \underline{\omega_c}, \overline{\omega_c}, C_m)$;
15: else return $(0, \underline{\omega_c}, \overline{\omega_c}, C_m)$;
```

which in return prunes fruitless seeds and makes S minimal. As illustrated as Case-1 and Case-2 in Fig. 2, it determines $\underline{\omega_c}$ and $\overline{\omega_c}$ for the next iteration if $S = \emptyset$. We deal with the case when $S \neq \emptyset$ in Line 9-15. When $S \neq \emptyset$, we invoke an exact algorithm *tciSeed* (Algorithm 5) to further condense seed set S (Line 10), which obtains new bounds of $\omega(s_i \cup \Lambda(s_i))$ for each seed $(s_i, \Lambda(s_i)) \in S$. In *tciSeed*, we use truss decomposition [46] and graph coloring to find an upper bound $\overline{\omega}_i$, and use an independent set of $\overline{\Lambda}(s_i)$ to acquire a lower bound $\underline{\omega}_i$. As proved in Theorem 7.3, for a given graph g, our novel independent set algorithm *compIS* (Algorithm 6) extracts a vertex subset of g that is included in one of the maximum independent sets of g, such a subset significantly reduces fruitless search space. Note that *tciSeed* updates $\underline{\omega_c}$, $\overline{\omega_c}$ as well as returns a minimal seed set S. When the improvement of the binary search is marginal (either $\overline{\omega_c} - \overline{\omega_c} < \theta$ or the bounds $\underline{\omega_c}$ and $\overline{\omega_c}$ cannot be improved), we invoke the algorithm *divSeed* (Algorithm 7) to find the maximum clique (Line 22) after further updating $\underline{\omega_c}$, $\overline{\omega_c}$, C_m, and a minimal seed set S using *scSeed* and *tciSeed* (Line 18-21). In brief, different from the previous algorithms that search the maximum clique directly, *divSeed* attempts to find a $(\underline{\omega_c} + 1)$-clique by generating several smaller/denser seeds from every seed $(s_i, \Lambda(s_i))$ in S iteratively. All these new smaller/denser seeds are further condensed, and a large percent of vertices/edges are pruned if they do not appear in the maximum clique. *divSeed* finds the final result when $S = \emptyset$. It is important to note that it offers us more power for pruning by finding a $(\underline{\omega_c} + 1)$-clique rather than finding the maximum clique directly.

7.1 Initialization

Algorithm *Init* (Algorithm 3) takes a graph G as input, and returns a 4-tuple $(r, \underline{\omega_c}, \overline{\omega_c}, C_m)$. Here, C_m is a clique, $\underline{\omega_c}$ and $\overline{\omega_c}$ are the lower and upper bounds, and r is an indicator whether C_m is the maximum clique. First, we compute the core number for every vertex u in G, denoted as core(u), using the core decomposition algorithm [5] (Line 1). Let the max core number be cm, and initialize $\overline{\omega_c}$ as $cm + 1$. If the max core is found as a clique, then the maximum clique is found, returning the result (Line 3-6). We discuss when the max core of G is not a clique (Line 7-15). We find the clique C_m among all the maximal cliques greedily found for every vertex v if its core number (core(v)) $\geq$ the current upper bound $\overline{\omega_c}$ (Line 7-10). If the current lower and upper bounds are the same, we return the result since C_m found is the maximum clique (Line 11). Next, we further update the current upper bound using graph coloring. Let the core number for G be cn (Line 12). $\overline{\omega_c}$ is reduced to cn if $\overline{\omega_c} > cn$. Finally we return the result. Note that at this stage, C_m found is the maximum clique if $\underline{\omega_c} = \overline{\omega_c}$.

7.2 The Sampling and the Core Reduction

We discuss the algorithm *scSeed* (Algorithm 4), which takes 2 inputs, a graph G and a specific k-clique value k, and returns a 4-tuple $(\underline{\omega_c}, \overline{\omega_c}, C_m, S)$, where $\underline{\omega_c}$ and $\overline{\omega_c}$ are a lower bound and an upper bound, and C_m is the current maximum clique, and S is a seed set. There are two main phases. The first phase is to generate an initial seed set S randomly (Line 1-6). The second phase is to update S using the reduction by k-core (Eq. (1)). We enlarge s and shrink $\Lambda(s)$ for each seed $(s, \Lambda(s))$ in S, and we will remove the entire $(s, \Lambda(s))$ from S if it cannot help to find a k-clique where k is given as an input of the algorithm (Line 7-33).

In the first phase, we sample a set edges from the edge set E of G, denoted as E_s, using uniform sampling. Then we construct the initial seed set S. For every seed, $(s, \Lambda(s))$, in S, s is an open triple $s = (u, v, w)$ if both (u, v) and (u, w) are in the sampled edge set E_s and (v, w) is in E. For each seed $(s, \Lambda(s))$, we determine its upper bound $\overline{\omega}_s$ as $\min\{\mathsf{core}(u), \mathsf{core}(v), \mathsf{core}(w)\}$ $+1$. We discuss the sampling probability p.

Theorem 7.1: *Let E_s be a subset of E of G by uniform sampling of edges from G with an edge sampling probability p. Let S_T be a set of open triangles, $s = (u, v, w)$, such that both (u, v) and (u, w) are in E_s and (v, w) is in E. Each k-clique in G contains at least one triangle in S_T with probability $\geq 1 - n^{-c}$, for $p = \sqrt{2 \cdot c \cdot \ln n/(k \cdot (k-1) \cdot (k-2))}$, where n is the number of vertices in G.*

Proof Sketch: Let $\mathcal{E}_1$ denote the event that one such specific open triangle is sampled from G. Then let $\mathcal{E}$ denote the event that at least one open triangle is sampled for a k-clique, and let $\overline{\mathcal{E}}$ denote the event $\mathcal{E}$ does not occur. We have

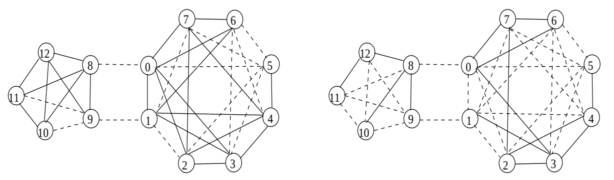
$$
\begin{aligned}
Pr(\mathcal{E}_1) &= p^2 = 2 \cdot c \cdot \ln n/(k \cdot (k-1) \cdot (k-2)) \\
Pr(\overline{\mathcal{E}}) &= (1 - Pr(E_1))^{k \cdot \binom{k-1}{2}} \leq n^{-c}
\end{aligned}
$$

Therefore, we can conclude $Pr(\mathcal{E}) \geq 1 - n^{-c}$. □

In the second phase, we update every seed $(s, \Lambda(s))$ in S and update S following the non-ascending order in the lower bounds ($\underline{\omega_s}$). It is important to note that the initial k in the loop (Line 7-33) is the input k of the algorithm for a k-clique and k may increase in order to prune more seeds from S. First, if $\overline{\omega}_s$ of a seed $(s, \Lambda(s))$ is less than k, the seed will be removed from S, since it cannot lead to a k-clique (Line 8). Second, we check if a k-clique can be found in $(s, \Lambda(s))$ by the condition of $|\Lambda(s)| + |s| < k$. If this condition does not hold, the seed will be removed from S (Line 9-11). Third, we further update a seed if the condition does not hold (Line 12-31). We discuss the third case below. Suppose we find $\Lambda(s)$ forms a clique, we update the the current maximum clique C_m by $(s, \Lambda(s))$, and update the current lower bound to be $\underline{\omega_c} = |C_m|$. It is worth noting that in every iteration k either remains unchanged or increases. Therefore, the C_m updated cannot be smaller than the one found before. We remove $(s, \Lambda(s))$ from S, since it is the current maximum clique. We follow Eq. (1), compute X_k, remove X_k from $\Lambda(s)$, and compute F (Line 16-18). Here, X_k is a subset of vertices in $\Lambda(s)$ where every vertex u has a core number which is less than $k - |s| - 1$. F is a subset of vertices in $\Lambda(s) \setminus X_k$ where every vertex connects to every other vertices in $\Lambda(s) \setminus X_k$. In other words, $s \cup F$ can possibly form a maximum clique. There are several cases. Case-i) When $\Lambda(s) \setminus F = \emptyset$: If $(s, \Lambda(s))$ cannot lead to a k-clique because F is empty, the seed $(s, \Lambda(s))$ will be removed from S (Line 19-21). Otherwise, if F is non-empty, the current maximum clique C_m can be updated by $s \cup F$, and the seed can be removed from S, since the seed is treated as the current maximum clique (Line 21-23).

Algorithm 4 *scSeed* (G, k)

```
1:  $S \leftarrow \emptyset$;
2:  let $E_s$ be a subset of $E$ where every $e \in E_s$ is sampled with probability $p$ from $E$;
3:  for each $u$ with $(u,v) \in E_s$, $(u,w) \in E_s$ and $(v,w) \in E$ do
4:      $s \leftarrow (u, v, w)$;
5:      $\overline{\omega}_s \leftarrow \min\{\mathsf{core}(u), \mathsf{core}(v), \mathsf{core}(w)\} + 1$, $S \leftarrow S \cup \{s\}$;
6:  end for
7:  for each $s \in S$ in non-ascending order of $\overline{\omega}_s$ do
8:      if $\overline{\omega}_s < k$ then $S \leftarrow S \setminus \{(s, \Lambda(s))\}$; continue;
9:      if $|\Lambda(s)| + |s| < k$ then
10:         $S \leftarrow S \setminus \{(s, \Lambda(s))\}$;
11:     else
12:         if $\Lambda(s)$ is a clique then
13:             $C_m \leftarrow s \cup \Lambda(s)$, $\underline{\omega_c} \leftarrow |C_m|$;
14:             $S \leftarrow S \setminus \{(s, \Lambda(s))\}$;
15:         else
16:             $X_k \leftarrow \{u \mid u \in \Lambda(s) \wedge \mathsf{core}(u) < k - |s| - 1\}$;
17:             compute $F$ from $\Lambda(s) \setminus X_k$;
18:             $\Lambda(s) \leftarrow \Lambda(s) \setminus X_k$;
19:             if $\Lambda(s) \setminus F = \emptyset$ and $F = \emptyset$ then
20:                 $S \leftarrow S \setminus \{(s, \Lambda(s))\}$;
21:             else if $\Lambda(s) \setminus F = \emptyset$ and $|F| + |s| \geq k$ then
22:                 $C_m \leftarrow s \cup F$, $\underline{\omega_c} \leftarrow |C_m|$;
23:                 $S \leftarrow S \setminus \{(s, \Lambda(s))\}$;
24:             else if $\Lambda(s) \setminus F \neq \emptyset$ and $|F| + |s| + 1 \geq k$ then
25:                 $C_m \leftarrow s \cup F \cup \{v\}$ where $v \in \Lambda(s) \setminus F$, $\underline{\omega_c} \leftarrow |C_m|$;
26:                 $(s, \Lambda(s)) \leftarrow (s \cup F, \Lambda(s) \setminus F)$;
27:             else
28:                 $(s, \Lambda(s)) \leftarrow (s \cup F, \Lambda(s) \setminus F)$;
29:             end if
30:         end if
31:         if $\underline{\omega_c} \geq k$ then $k \leftarrow \underline{\omega_c} + 1$;
32:     end if
33: end for
34: let $\overline{\omega_c}$ be the core number of the max core found for non-empty seeds;
35: return $(\underline{\omega_c}, \overline{\omega_c}, C_m, S)$;
```

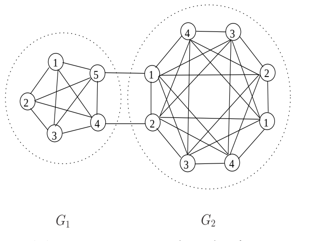


(a) For a 4-clique (b) For a 5-clique

Figure 6: Sampling edges for finding a ω_t-clique

Case-ii) when $\Lambda(s) \setminus F \neq \emptyset$: it will enlarge s by $s \cup F$ and shrink $\Lambda(s)$ by $\Lambda(s) \setminus F$. In addition, if $|s \cup F| + 1 \geq k$, it implies a k-clique has been found (Line 24), and we will update the current maximum clique by a subgraph with $s \cup F \cup \{v\}$ where v is taken from $\Lambda(s) \setminus F$ (Line 25). Note that the lower bound $\underline{\omega_c}$ is updated when the current maximum clique is updated. After consideration of the cases, we update k to be $\underline{\omega_c} + 1$ if $\underline{\omega_c} \geq k$ (Line 31). Finally, we return $(\underline{\omega_c}, \overline{\omega_c}, C_m, S)$ where $\overline{\omega_c}$ is the core number of the max core found for non-empty seeds.

Example 7.1: Reconsider the graph G in Fig. 1. Fig. 6(a) and Fig. 6(b) illustrate the sampled edges, which are in solid lines, when *scSeed* attempts to find a 4-clique and 5-clique, respectively. Here, $c = 2$ s.t. each ω_t-clique will be missing with probability $< n^{-2}$. In Fig. 6(a), each edge is sampled with probability $p = 0.462$. *scSeed* extracts all seeds which can be a seed for a 4-clique. For instance, $(s_1, \Lambda(s_1))$ is such a seed, where $s_1 = \{0, 1, 2\}$ and $\Lambda(s_1) = \{3, 7\}$. For this seed, the lower bound is a 4-clique since s_1 has common neighbors in $\Lambda(s_1)$, and the upper bound is a 4-clique, because $\Lambda(s_1)$ is a 0-core from which it cannot get a large clique. The maximum clique containing s_1 is a 4-clique. The open triangle $s_2 = \{0, 1, 4\}$ cannot be a valid seed since


(a) By truss and coloring

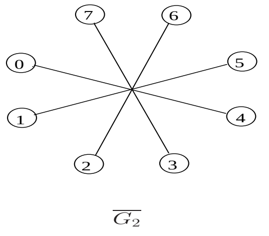
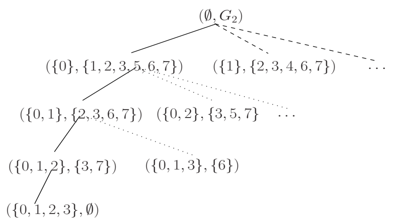

(b) By independent set

Figure 7: Reduction by TCI


(a) By the existing *DFS* tree T_d

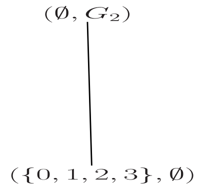

(b) By our new *BFS* tree, T_b

Figure 8: *DFS* T_d **vs** *BFS* T_b

$(0, 4) \notin E$. The seed $(s_3, \Lambda(s_3))$ is a seed where $s_3 = \{8, 9, 10\}$ and $\Lambda(s_3) = \{11, 12\}$. Since $\Lambda(s_3)$ itself is a clique, the maximum clique containing s_3 is a 5-clique. In Fig. 6(b), each edge is sampled with probability $p = 0.414$. Consider the seed $(s_4, \Lambda(s_4))$ where $s_4 = \{0, 1, 3\}$ and $\Lambda(s_4) = \{2, 6\}$. s_4 cannot be included in a 5-clique, since $\Lambda(s_4)$ is a 0-core which cannot be a 2-cliques.

7.3 The Reduction by TCI

We discuss the reduction by k-truss, coloring, and independent set following Eq. (2). The algorithm *tciSeed* is shown in Algorithm 5, which takes 3 inputs, a graph G, a seed set S, and a specific k-clique value k, and returns a 4-tuple $(\underline{\omega_c}, \overline{\omega_c}, C_m, S)$, where $\underline{\omega_c}$ and $\overline{\omega_c}$ are a lower bound and an upper bound, and C_m is the current maximum clique, and S is a seed set reduced.

As given in Eq. (2), in *tciSeed*, for each seed $(s, \Lambda(s)) \in S$, we shrink $\Lambda(s)$ to be $\Lambda(s) \setminus (X_t \cup X_c)$, where both X_t and X_c are two subsets in $\Lambda(s)$ such that any vertex in $X_t \cup X_c$ cannot appear in a k-clique. Here X_t contains the vertices that cannot be in a $(\omega_t - |s|)$-truss in $\Lambda(s)$, and X_c contains vertices whose neighbors are colored with less than $\omega_t - |s| - 1$ colors. The correctness of X_t is obvious. We prove the correctness of X_c below.

Theorem 7.2: *Given a graph G with a graph coloring, if $\omega(G) \geq k$, then any vertex v with neighbors colored with less than $k - 1$ colors cannot be included in any k-clique of G.*

Proof Sketch: Assume the opposite, i.e., there is a vertex v whose neighbors can be colored with less than $k - 1$ colors while v itself is included in a k-clique. Then, the subgraph induced by $\{v\} \cup \Gamma(v)$ contains a k-clique and can be colored with less than k colors, which leads to a contradiction. □

Next, we discuss how to enlarge s in the seed of $(s, \Lambda(s))$ by a subset $P \subseteq \Lambda(s)$, where all vertices in P belong to a maximum clique in $\Lambda(s)$. We find P as the vertices in a maximum independent set in the complementary graph $\overline{\Lambda(s)}$ in an algorithm *compIS*, based on Theorem 7.3.

Theorem 7.3: *In a graph G, if there is a vertex v with degree $\delta(v) = 1$, there is at least one maximum independent set of G containing v in G.*

Proof Sketch: Assume the opposite, i.e., none of the maximum independent sets contains v in G. Suppose I is such a maximum independent set in G. Let the unique neighbor of v be u. There are only two cases regarding the existence of u in I. First, if $u \in I$, we can have a maximum independent set I' by replacing u with v such that $I' \leftarrow I \cup \{v\} \setminus \{u\}$. Second, if $u \notin I$, we can have a larger independent set I' by enlarging I to $I \cup \{v\}$ such that $I' \leftarrow I \cup \{v\}$. This leads to the conclusion that at least one maximum independent set contains v. □

The algorithm *compIS* is given in Algorithm 6 to compute an independent set I for an input graph G and a set of vertices $P \subseteq I$ in G, where every vertex in P must appear in a maximum independent set of G. We find an independent set I in G by selecting vertices in the non-descending order of their degrees in G. When a vertex, v is added into I, we remove its edges to its neighbors u in G as well as the edges of its neighbors, and reduce the degree of w by one if there is an edge (u, w) in G. The graph after such edges removal is the graph to be processed next, which we call a remaining graph. We find P based on Theorem 7.3 by adding a vertex whose degree is 1 into P repeatedly until all vertices in the remaining graph have degrees greater than 1.

We explain *tciSeed* (Algorithm 5). For every seed $(s, \Lambda(s))$ in non-ascending order of $\overline{\omega_c}$ and $\underline{\omega_c}$, we reduce $\Lambda(s)$ by k-truss and coloring, and enlarge s by the independent set. First, we compute $((k\text{-}|s|))$-truss and X_t where X_t is a subset of $\Lambda(s)$ excluded from the $(k\text{-}|s|)$-truss of $\Lambda(s)$. We reduce $\Lambda(s)$ by $\Lambda(s) \setminus X_t$ (Line 4-5). If the $(k\text{-}|s|)$-truss is a clique, we treat $s \cup \Lambda(s)$ as the current maximum clique, and remove the seed of $(s, \Lambda(s))$ from S (Line 6-8). Second, we compute graph coloring for $\Lambda(s)$ and X_c where X_c is a subset of $\Lambda(s)$ whose neighbors can be colored with $< k - |s| - 1$ colors. We reduce $\Lambda(s)$ by $\Lambda(s) \setminus X_c$ (Line 9-11). In addition, if the remaining $\Lambda(s)$ is empty, the seed of $(s, \Lambda(s))$ can be removed from S (Line 12). Third, we enlarge s by $s \cup P$, where P is a subset of vertices in $\Lambda(s)$ by the algorithm *compIS* as discussed (Line 13-14). Note that *compIS* also returns an independent set, I, for $\overline{\Lambda(s)}$. If $|I| + |s| \geq k$, we treat $s \cup I$ as the current maximum clique (Line 15).

Example 7.2: Suppose the graph G in Fig. 1 is $\Lambda(s)$ for some seed s. We show how to find a k-clique in G. Fig. 7(a) illustrates the idea by truss and coloring, where the number on a vertex is the color number of the vertex. Let $k = 6$, G_1 is pruned since edges in G_1 are with truss value 5. Let $k = 5$, G_2 can be pruned safely, since their neighbors can be colored with less than 4 colors. We demonstrate the idea of independent set by *compIS* in Fig. 7(b). Consider G_2. Here, each vertex in the complementary graph $\overline{G_2}$ is with degree 1, then *compIS* finds an independent set $I = \{0, 1, 2, 3\}$. Because $P = I$, P is the maximum independent set of $\overline{G_2}$. In other words, the maximum clique of G_2 is a 4-clique. Furthermore, consider finding the maximum clique of G_2. Search trees T_d and T_b are outlined in Fig. 8(a) and Fig. 8(b), respectively. As can be seen, T_b is better than T_d. First, T_b prunes fruitless branches, in dashed lines starting from the root $(\emptyset, G_2)$. Second, T_b reduces branches along the path from $(\emptyset, G_2)$ to $(\{0, 1, 2, 3\}, \emptyset)$, which are in dotted lines in Fig. 8(a).

7.4 The Reduction by Dividing

The k-clique decision problem is NP-complete. Therefore, there are cases where the brute-force search is unavoidable. In our algorithm, such cases are when S is non-empty. By the existing brute-force search, for each seed $(s, \Lambda(s))$ in S, it branches from each vertex $v \in \Lambda(s)$ to enumerate maximal cliques and prune fruitless

Algorithm 5 *tciSeed* (G, S, k)

```
1: for each $(s, \Lambda(s)) \in S$ do $\overline{\omega}_s \leftarrow |s| + |\Lambda(s)| - 1, \underline{\omega}_s \leftarrow |s| + 1; \underline{\omega}_s$;
2: for each $(s, \Lambda(s)) \in S$ in non-ascending order of $\overline{\omega}_s$ and $\underline{\omega}_s$ do
3:     if $\overline{\omega}_s < k$ then $S \leftarrow S \setminus \{(s, \Lambda(s))\}$;
4:     compute the $(k$-$|s|)$-truss from $\Lambda(s)$ and $X_t$;
5:     $\Lambda(s) \leftarrow \Lambda(s) \setminus X_t$;
6:     if the $(k$-$|s|)$-truss is a clique then
7:         $C_m \leftarrow s \cup \Lambda(s), \underline{\omega_c} \leftarrow |C_m|$;
8:         $S \leftarrow S \setminus \{(s, \Lambda(s))\}$;
9:     else if $\Lambda(s) \neq \emptyset$ then
10:        compute graph coloring for $\Lambda(s)$ and $X_c$;
11:        $\Lambda(s) \leftarrow \Lambda(s) \setminus X_c$;
12:        if $\Lambda(s) = \emptyset$ then $S \leftarrow S \setminus \{(s, \Lambda(s))\}$;
13:        $(P, I) \leftarrow$ compIS $(\overline{\Lambda(s)})$;
14:        $(s, \Lambda(s)) \leftarrow (s \cup P, \Lambda(s) \cap \Lambda(P))$;
15:        if $|I| + |s| \geq k$ then $C_m \leftarrow s \cup I, \underline{\omega_c} \leftarrow |C_m|$;
16:    end if
17:    if $\underline{\omega_c} \geq k$ then $k \leftarrow \underline{\omega_c} + 1$;
18: end for
19: return $(\underline{\omega_c}, \overline{\omega_c}, C_m, S)$;
```

Algorithm 6 *compIS* $(G = (V, E))$

```
1: $P \leftarrow \emptyset, I \leftarrow \emptyset$;
2: add every $(v, \delta(v))$ into a min-heap $H$ for $v \in V$;
3: initialize $t$ to be $true$, and $state[1...|V|]$ be all zeros;
4: while $H \neq \emptyset$ do
5:     $v \leftarrow$ getMin($H$);
6:     if $state[v] = 0$ then
7:         $state[v] \leftarrow 1, I \leftarrow I \cup \{v\}$;
8:         if $\delta(v) > 1$ then $t \leftarrow false$;
9:         if $t = true$ then $P \leftarrow P \cup \{v\}$;
10:        for every $u$ with $(u, v) \in G$ and $state[u] = 0$ do
11:            $state[u] \leftarrow 1$;
12:            for each $w$ with $(u, w) \in G$ and $state[w] = 0$ do
13:                $\delta(w) \leftarrow \delta(w) - 1$;
14:            end for
15:        end for
16:    end if
17: end while
18: return $(P, I)$;
```

branches. In this paper, we propose a new approach to significantly improve the performance of the existing brute-force search. As indicated in *RMC* (Algorithm 2) in Line 22, we find the maximum clique from the seed set S by calling the algorithm *divSeed* until S becomes empty.

The Algorithm *divSeed* is given in Algorithm 7, which takes 3 inputs: the graph G, the seed set S, and a k value for specific k-clique. Such a k value offers more chances for *divSeed* to prune. Like *scSeed* and *tciSeed*, *divSeed* returns a 4-tuple $(\underline{\omega_c}, \overline{\omega_c}, C_m, S)$. Unlike *scSeed* and *tciSeed*, *divSeed* returns a seed set, denoted as S', to replace the input S. Here, a seed $(s, \Lambda(s)) \in S$ is replaced by several smaller and denser seeds $(s', \Lambda(s'))$ in S', as illustrated in Fig. 4(c). For each seed $(s, \Lambda(s))$ in S, we compute V_1 and V_2. As discussed, V_1 is a subset of $\Lambda(s)$ where every vertex in V_1 has a color greater than or equal to $k - |s|$, and V_2 is a subset of $\Lambda(s)$ where for every vertex in V_2 its neighbors in $\Lambda(s)$ can be colored with less than $k - |s| - 1$ colors (Line 3-4). By Theorem 7.2, vertices in V_2 cannot be in a k-clique. We remove the vertices in V_2 from $\Lambda(s)$, and refine V_1 to be $V_1 \setminus V_2$ (Line 5). We show that any k-clique of $s \cup \Lambda(s)$ must contain at least one vertex from a subset $V_1 \subseteq \Lambda(s)$ Theorem 7.4.

Theorem 7.4: *For a graph G, if $\omega(G) \geq k$, then any k-clique of G must contain at least one vertex with color $\geq k$.*

Proof Sketch: We show the claim by contradiction, i.e., we assume that there is a k-clique with all vertices colored $< k$ in G. Then, the coloring is still valid if we remove all the other vertices that are not included in the k-clique. This implies that we get a coloring for a k-clique that uses $< k$ colors, leading to a contradiction. Therefore, any k-clique in G contain at least one vertex with color $\geq k$. □

Algorithm 7 *divSeed* (G, S, k)

```
1: $S' \leftarrow \emptyset$;
2: for each $(s, \Lambda(s)) \in S$ do
3:     $V_1 \leftarrow \{u \mid u \in \Lambda(s) \wedge \mathsf{color}(u) \geq k - |s|\}$;
4:     $V_2 \leftarrow$ the vertices in $\Lambda(s)$ whose neighbors in $\Lambda(s)$ can be colored with
       $< k - |s| - 1$ colors;
5:     $\Lambda(s) \leftarrow \Lambda(s) \setminus V_2, V_1 \leftarrow V_1 \setminus V_2$;
6:     for each $v \in V_1$ do
7:         $s' \leftarrow s \cup \{v\}, \Lambda(s') \leftarrow \Lambda(s) \cap \Gamma(v)$;
8:         if $|s'| + |\Lambda(s')| < k$ then continue;
9:         if $\Lambda(s')$ is a clique then
10:            $C_m \leftarrow s' \cup \Lambda(s'), \underline{\omega_c} \leftarrow |C_m|$;
11:        else
12:            $X_k \leftarrow \{u \mid u \in \Lambda(s') \wedge \mathsf{core}(u) < k - |s'| - 1\}$;
13:            compute $F$ from $\Lambda(s') \setminus X_k$;
14:            $\Lambda(s') \leftarrow \Lambda(s') \setminus X_k$;
15:            if $\Lambda(v') \setminus F = \emptyset$ then
16:                if $|F| \geq k - |s'|$ then $C_m \leftarrow s' \cup F, \underline{\omega_c} \leftarrow |C_m|$;
17:            else
18:                if $|F| \geq k - |s'| - 1$ then
19:                    $C_m \leftarrow s' \cup F \cup u$ where $u \in \Lambda(v) \setminus F$;
20:                    $\underline{\omega_c} \leftarrow |C_m|$;
21:                    $s' \leftarrow s' \cup F, S' \leftarrow S' \cup (s', \Lambda(s'))$;
22:                else
23:                    $s' \leftarrow s' \cup F, S' \leftarrow S' \cup (s', \Lambda(s'))$;
24:                end if
25:            end if
26:        end if
27:        if $\underline{\omega_c} \geq k$ then $k \leftarrow \underline{\omega_c} + 1$;
28:    end for
29: end for
30: $(\underline{\omega_c}, \overline{\omega_c}, C_m, S) \leftarrow tciSeed(G, S', k)$;
31: return ;
```

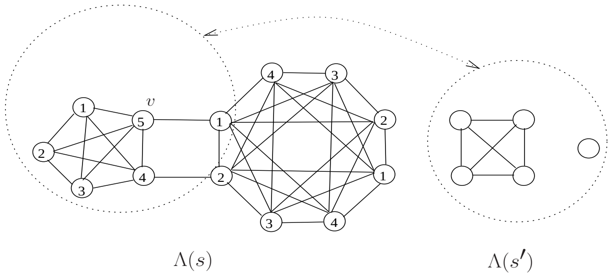


Figure 9: The reduction by dividing

In *divSeed*, in the loop (Line 6-28), we get a new smaller/denser seed, $(s', \Lambda(s'))$ from the current seed $(s, \Lambda(s))$ for every vertex v in V_1. Here, s' becomes $s \cup \{v\}$, and $\Lambda(s')$ becomes $\Lambda(s) \cap \Gamma(v)$ (Line 7). Then, we reduce the new seed $(s', \Lambda(s')$ using the similar techniques used in the *tciSeed* algorithm (Line 6-28).

Example 7.3: Suppose the graph G in Fig. 1 is $\Lambda(s)$ for some s, and we search for the maximum clique in $\Lambda(s)$. In Fig. 9, the left is $\Lambda(s)$ where the number of a vertex is its color. Since $\Lambda(s)$ is colored with 5 colors, we have $\omega(\Lambda(s)) \leq 5$, which implies $k \leq 5$. We can get a smaller/denser seed $(s', \Lambda(s'))$ from $(s, \Lambda(s))$ by the *divSeed* algorithm. Here, s' becomes $s \cup \{v\}$ where v is the vertex with the max color number 5 in $\Lambda(s)$ as shown in the left in Fig. 9, $\Lambda(s')$ becomes $\Lambda(s) \cap \Gamma(s')$ as shown in the right half of Fig. 9. Suppose $k = 5$, finding a 5-clique in $\Lambda(s)$ is equivalent to finding a 4-clique in $\Lambda(s')$.

8. EXPERIMENTAL STUDIES

We have conducted experimental studies using 22 real large graphs to compare *RMC* with several state-of-the-art approaches, including *FMC* [37], *PMC* [40], *MCQD* [25], *cliquer* [34], which are exact algorithms, and *GRASP* [1], which is an approximation algorithm. For *PMC*, we set the number of threads to be one. For *MCQD*, among all variants, we report results on *MCQD* +CS (which utilizes improved coloring and dynamic sorting), since it is the best-performing variant. For *GRASP*, we repeat 1000 times and select the largest clique found, following [1]. All the algorithms are implemented in C++ and complied by gcc 4.8.2, and tested on machine with 3.40GHz Intel Core i7-4770 CPU, 32GB RAM and running Linux. The time unit used is second and we set time limit as 24 hours. For *RMC*, we set $c = 0.2$ (Theorem 7.1) and $\theta = 3$ (In Algorithm 2, Line 5).

Datasets: We use 22 real-world networks and 2 series of synthetic graphs. Among the real-world networks, Epinions, LiveJournal, Pokec, Slashdot0811, Slashdot0902, wikivote, Youtube, Orkut, BuzzNet, Delicious, Digg, Flixster, Foursquare and Friendster are social networks; BerkStan, Google, NotreDame, Stanford are web graphs; Gnutella is a peer-to-peer network; Amazon is a product co-purchasing network; WikiTalk is a communication network and lastfm is a music website. Amazon, BerkStan, Epinions, Gnutella, Google, LiveJournal, NotreDame, Pokec, Slashdot0811, Slashdot0902, Stanford, WikiTalk and wikivote are downloaded from Stanford large network dataset collection (`http://snap.stanford.edu/data`); Youtube and Orkut are from `http://socialnetworks.mpi-sws.org/datasets.html`; BuzzNet, Delicious, Digg, Flixster, Foursquare, Friendster and Lastfm are downloaded from `http://socialcomputing.asu.edu/pages/datasets`. The detailed information of the real-world datasets are summarized in Table 1. In the table, for each graph, the 2nd and 3rd columns show the numbers of vertices and edges[1], respectively. The 4th, 5th, 6th and 7th columns show the size of max degree, max core value, max truss value and the clique number of each graph. For synthetic graphs, we use SNAP library [29] to generate networks by PowerLaw (PL) [33] with n vertices and exponent α, and by Watts-Strogatz (WS) [48] with n vertices where each vertex is connected to k nearest neighbors in the ring topology and each edge is rewired with probability p. We do not consider Erdős-Rényi random graph model [12] since the graphs are significantly different from real-world massive graphs, and we neglect Barabási-Albert preferential attachment model [4] since the maximum clique is known to be a $(k + 1)$-clique, where k is the number of edges each new vertex link to existing vertices.

The efficiency on real datasets: Table 2 shows the efficiency of *RMC* and its comparisons. The 12th and 15th columns show the sizes of the maximum cliques found by *GRASP* and *RMC*, respectively. We neglect the results of other algorithms since they are exact algorithms. According to the 7th column of Table 1, *RMC* finds the maximum cliques in all real graphs while *GRASP* cannot in most graphs. The largest cliques found by *GRASP* in some graphs (e.g., BerkStan and LiveJouranl) are much smaller than the optimal solutions. The 2nd, 4th, 6th, 8th, 10th and 13th columns show the running time of *FMC*, *PMC*, *MCQD*, *cliquer*, *GRASP* and *RMC*, respectively. In Table 2, the symbol of "-" indicates that the algorithm cannot get the result in 24 hours. *RMC* outperforms others in 19 out of 22 datasets. The advantage of *RMC* on graphs like Orkut is significant. Only *FMC* and *PMC* outperform *RMC* on 3 datasets (Amazon, Gnutella and NotreDame) marginally. The 3rd, 5th, 7th, 9th, 11th and 14th columns show the memory consumption of *FMC*, *PMC*, *MCQD*, *cliquer*, *GRASP* and *RMC*, respectively. As demonstrated, *RMC* outperforms all others in all datasets tested significantly. From Table 2, *RMC*, *PMC* and *FMC* outperform *MCQD*, *cliquer* and *GRASP* significantly in most graphs, therefore, we will focus on the three algorithms below.

[1]for each dataset, we remove directions if included and delete all self-loops and multi-edges if exist.

Table 1: Summarization of datasets

Graph	$\|V\|$	$\|E\|$	d_{max}	c_{max}	t_{max}	$\|\omega(G)\|$
Amazon	403,394	2,443,408	2,752	10	11	11
BerkStan	685,230	6,649,470	84,230	201	201	201
Epinions	75,879	405,740	3,044	67	33	23
Gnutella	62,586	147,892	95	6	4	4
Google	875,713	4,322,051	6,332	44	44	44
LiveJournal	4,036,538	34,681,189	14,815	360	352	327
NotreDame	325,729	1,090,108	10,721	155	155	155
Pokec	1,632,803	22,301,964	14,854	47	29	29
Slashdot0811	77,360	469,180	2,539	54	35	26
Slashdot0902	82,168	504,229	2,552	55	36	27
Stanford	281,903	1,992,636	38,625	71	62	61
WikiTalk	2,394,385	4,659,563	100,029	131	53	26
wikivote	7,115	100,762	1,065	53	23	17
Youtube	1,138,499	2,990,443	28,754	51	19	17
Orkut	3,072,627	117,185,083	33,313	253	78	51
BuzzNet	101,163	2,763,066	64,289	153	59	31
Delicious	536,408	1,366,136	3,216	33	23	21
Digg	771,229	5,907,413	17,643	236	73	50
Flixster	2,523,386	7,918,801	1,474	68	47	31
Foursquare	639,014	3,214,986	106,218	63	38	30
Friendster	5,689,498	14,067,887	4,423	38	31	25
Lastfm	1,191,812	4,519,340	5,150	70	23	14

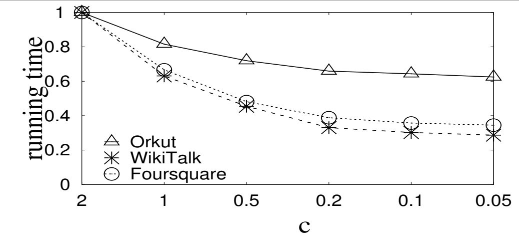


Figure 10: The Efficiency of RMC: Varying c

The effectiveness of the binary search and divSeed: In order to illustrate the effectiveness of the binary search and *divSeed*, we show the lower bound $\underline{\omega_c}$ and upper bound $\overline{\omega_c}$ returned by the binary search, and the time needed for *divSeed* (the column of $time_{div}$ in Table 2) to determine the maximum clique in the 17th, 18th and 16th columns. First, in a large percent of graphs tested, for instance, Amazon, BerkStan, Google and Pokec, there is no need to invoke *divSeed*. Even for graphs that need *divSeed* to determine the maximum clique, the time spent by *divSeed* is much less than expected. This is largely due to the excellent pruning power of the binary search. Second, consider those graphs that need invoke *divSeed*, in a large percent of these graph, the lower bound $\underline{\omega_c}$ is actually the optimal $\omega(G)$.

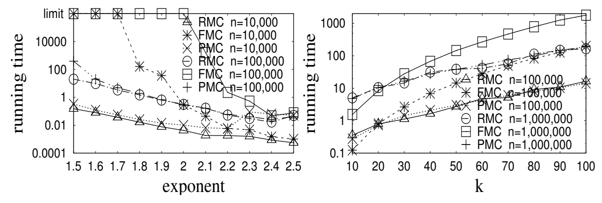
The influence of the parameter c: We show the influence of c in Theorem 7.1 on the accuracy and efficiency of *RMC*. Here we show the results using 3 datasets: Orkut, WikiTalk, and Foursquare. The remaining datasets are similar to one of the three. Fig. 10 demonstrates the running time by employing different values of c. Here, we set the running time as 1 when $c = 2$, and use the ratio to indicate the running time with smaller c. As can be seen, 1) employing smaller c improves the performance of *RMC* significantly, 2) the improvement is marginal when c is small enough, and 3) the improvement differs significantly in different graphs. The influence of c on the accuracy of *RMC* is shown in Table 3. There are some differences in the 3 graphs. 1) For Orkut, *RMC* finds the maximum clique even when $c = 0.05$ and the bounds $\underline{\omega_c}$ and $\overline{\omega_c}$ returned by the binary search are the same for different c. 2) For WikiTalk, *RMC* always returns the correct result while the bounds $\underline{\omega_c}$ and $\overline{\omega_c}$ returned by smaller c is better than the bounds obtained when

Table 2: Performance of *FMC*, *PMC*, *MCQD*, *cliquer*, *GRASP* and *RMC* on real-world networks

Graph	FMC		PMC		MCQD		cliquer		GRASP			RMC					
	time	mem	time	mem	time	mem	time	mem	time	mem	$\omega(G)$	time	mem	$\omega(G)$	t_{div}	$\underline{\omega_c}$	$\overline{\omega_c}$
Amazon	0.1	112	**0.09**	114	-	-	868.1	19,437	63.17	85	9	0.1	67	11	0	11	11
BerkStan	-	-	1.04	272	-	-	-	-	708.5	224	53	**0.23**	144	201	0	201	201
Epinions	5.65	15	0.26	46	7.33	647	31.81	696	15.68	18	22	**0.11**	12	23	0.02	23	26
Gnutella	**0.01**	8	0.05	8	4.37	10	22.52	470	2.4	8	4	0.03	8	4	0.01	4	6
Google	0.35	216	0.77	250	-	-	-	-	80.61	163	20	**0.24**	135	44	0	44	44
LiveJournal	-	-	6.15	1,412	-	-	-	-	508.49	995	102	**1.82**	788	327	0.01	326	327
NotreDame	**0.01**	215	0.19	474	-	-	449.73	12,857	30.01	61	154	0.2	46	155	0	155	155
Pokec	-	-	9.51	1,359	-	-	-	-	369.18	575	13	**2.79**	407	29	0	29	29
Slashdot0811	6.66	21	0.18	38	7.47	1,943	33.24	724	23.26	18	24	**0.06**	13	26	0.01	25	32
Slashdot0902	10.18	18	0.19	36	8.53	1,936	37.5	817	14.45	19	25	**0.07**	14	27	0.01	27	29
Stanford	-	-	0.48	68	-	-	428.31	9,550	201.31	85	49	**0.19**	51	61	0	61	61
WikiTalk	2,767.95	325	4.87	336	-	-	-	-	298.42	415	24	**2.45**	314	26	0.41	22	37
wikivote	1.53	3	0.08	49	0.06	3	0.29	435	2.59	3	17	**0.03**	2	17	0.01	17	21
Youtube	6.13	187	2.04	191	-	-	-	-	1,145.82	216	15	**1.26**	156	17	0.08	17	20
Orkut	-	-	208.97	1,692	-	-	-	-	9,132.23	2,520	25	**45.01**	1,623	51	3.76	50	65
BuzzNet	28,354.4	88	14.37	52	13.15	1479	51.35	1236	758.99	81	30	**4.78**	48	31	1.89	23	59
Delicious	0.27	83	0.26	86	-	-	1,476.15	31,987	33.14	103	20	**0.1**	73	21	0.01	20	21
Digg	-	-	10.09	1,388	-	-	-	-	593.21	200	44	**2.13**	146	50	0.38	45	58
Flixster	71.10	434	2.25	423	-	-	-	-	120.52	473	26	**1.04**	358	31	0.06	31	33
Foursquare	-	-	39.36	141	-	-	13,698.37	-	470.83	153	27	**23.42**	113	30	1.82	29	33
Friendster	10.12	823	2.88	852	-	-	-	-	254.74	939	14	**1.09**	765	25	0	25	25
Lastfm	25.61	205	2.55	265	-	-	-	-	80.84	243	14	**1.41**	177	14	0.2	14	17

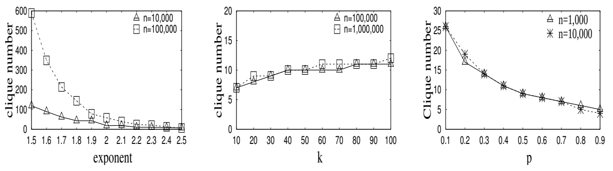
Table 3: The accuracy of RMC: Varying c

c	Orkut			WikiTalk			Foursquare		
	$\underline{\omega_c}$	$\overline{\omega_c}$	$\omega(G)$	$\underline{\omega_c}$	$\overline{\omega_c}$	$\omega(G)$	$\underline{\omega_c}$	$\overline{\omega_c}$	$\omega(G)$
2	50	65	51	22	55	26	29	33	30
1	50	65	51	22	37	26	29	33	30
0.5	50	65	51	22	37	26	29	33	30
0.2	50	65	51	22	37	26	29	33	30
0.1	50	65	51	22	37	26	29	33	29
0.05	50	65	51	22	37	26	29	33	29



(a) PowerLaw graph (b) Watts-Strogatz model

Figure 11: *FMC*, *PMC* and *RMC* on synthetic graphs



(a) PowerLaw graph (b) WS model (c) WS model

Figure 12: Clique size in synthetic graphs

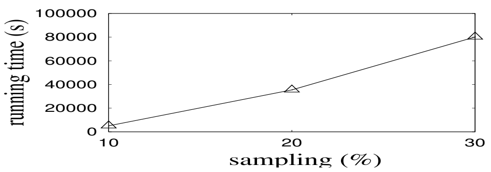


Figure 13: Scalability of *RMC* on Twitter

$c = 2$. 3) For Foursquare, *RMC* misses the maximum clique when $c = 0.1$ and $c = 0.05$. On one hand, a smaller c improves the efficiency and the bounds $\underline{\omega_c}$ and $\overline{\omega_c}$ as it generates fewer seeds. On the other hand, it may introduce some errors.

The efficiency on synthetic datasets: We compare *RMC*, *FMC* and *PMC* on synthetic graphs by PL [33] and WS [48] (Fig. 11). *RMC* and *PMC* significantly outperforms *FMC* in both PL graphs and WS graphs. The improvement of *RMC* over *PMC* is marginal. We also study how each parameter in the models influences $\omega(G)$. From Fig. 12(a) and Fig. 12(b), it is interesting to find that, in PL graphs, $\omega(G)$ increases significantly as n increases and α decreases. However, although k has a positive correlation with $\omega(G)$ in WS, the influence is not as obvious as that in PL graphs. In addition, the effect of n in WS is negligible. The influence of rewiring probability p of WS is shown in Fig. 12(c). As can be seen, the clique number decreases as p increases. When p approaches 1, the graph approximates an Erdős-Rényi random graph [12].

The effectiveness of reduction in *RMC*: We study the effectiveness of the reduction techniques used in *RMC*, namely k-core, coloring, the independent set, and k-truss. We take *RMC* with all the 4 reduction techniques as the baseline (unit 1), and test the effectiveness of a reduction technique by disabling it, using all the 22 datasets. The average slowdown by disabling k-core, coloring, the independent set, and k-truss are 79.17%, 39.84%, 80.62%, and -5.27%, respectively. The reason for k-truss to be -5.27% on average is because k-core has pruned almost all the cases. In addition, the maximum slowdown by disabling k-core, coloring, the independent set, and k-truss are 869.38%, 141.44%, 1,033.88%, and 17.32%. The reduction techniques are effective in improving the *RMC* performance.

The scalability of *RMC*: We study the scalability of *RMC* using a Twitter social graph with 41,652,230 nodes and 1,202,513,046 edges [27]. To test the scalability, we sample three Twitter subgraphs with 10%, 20% and 30% edges. The numbers of nodes and edges for the 3 sampled Twitter subgraphs are 27,640,227 and 120,243,815, 33,675,404 and 240,490,238, and 36,653,883 and 360,757,098, respectively. Fig. 13 shows that *RMC* scales linearly.

9. CONCLUSION

In this paper, we study the maximum clique problem. Different from previous algorithms, our approach *RMC* employs a binary search schema. Specifically, *RMC* maintains a lower bound $\underline{\omega_c}$ and an upper bound $\overline{\omega_c}$ of $\omega(G)$ and attempts to find a ω_t-clique in each iteration where $\omega_t = \lfloor(\underline{\omega_c} + \overline{\omega_c})/2\rfloor$. Due to the NP-completeness of finding a ω_t-clique in each iteration, *RMC* samples a seed set S s.t. finding a ω_t-clique in graph G is equivalent to finding a ω_t-clique in S with probability guarantees ($\geq 1 - n^{-c}$). For the remaining seeds whose maximum cliques can probably update the maximum clique C_m found so far, we propose a new iterative brute-force searching algorithm *divSeed* to find the maximum clique when the seed set S becomes empty. Our extensive experimental studies demonstrate the efficiency and robustness of our approach. First, *RMC* finds the exact maximum cliques in all datasets tested. Second, in 19 out of 22 datasets, *RMC* outperforms others, and the performance difference in the other 3 datasets is marginal.

ACKNOWLEDGEMENTS: This work was supported by the grant of the Research Grants Council of the Hong Kong SAR, China, No. 14221716.

10. REFERENCES

[1] J. Abello, M. G. Resende, and S. Sudarsky. Massive quasi-clique detection. In *Latin American Symposium on Theoretical Informatics*, pages 598–612. Springer, 2002.

[2] T. Akiba, Y. Iwata, and Y. Yoshida. Linear-time enumeration of maximal k-edge-connected subgraphs in large networks by random contraction. In *Proc. of CIKM*, 2013.

[3] E. Akkoyunlu. The enumeration of maximal cliques of large graphs. *SIAM Journal on Computing*, 2(1):1–6, 1973.

[4] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439), 1999.

[5] V. Batagelj and M. Zaversnik. An o (m) algorithm for cores decomposition of networks. *arXiv preprint cs/0310049*, 2003.

[6] N. Berry, T. Ko, T. Moy, J. Smrcka, J. Turnley, and B. Wu. Emergent clique formation in terrorist recruitment. In *Prof. of AAAI-04 Workshop on Agent Organizations: Theory and Practice*, 2004.

[7] C. Bron and J. Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Communications of the ACM*, 16(9), 1973.

[8] R. Carraghan and P. M. Pardalos. An exact algorithm for the maximum clique problem. *Operations Research Letters*, 9, 1990.

[9] J. Cheng, Y. Ke, A. W.-C. Fu, J. X. Yu, and L. Zhu. Finding maximal cliques in massive networks by h*-graph. In *Proc. of SIGMOD*, pages 447–458, 2010.

[10] J. Cheng, L. Zhu, Y. Ke, and S. Chu. Fast algorithms for maximal clique enumeration with limited memory. In *Proc. of SIGKDD*, 2012.

[11] S. A. Cook. The complexity of theorem-proving procedures. In *Proc. of STOC*, 1971.

[12] P. Erdös and A. Rényi. On random graphs i. *Publ. Math. Debrecen*, 6, 1959.

[13] T. Fahle. Simple and fast: Improving a branch-and-bound algorithm for maximum clique. In *European Symposium on Algorithms*, 2002.

[14] U. Feige. Approximating maximum clique by removing subgraphs. *SIAM Journal on Discrete Mathematics*, 18, 2004.

[15] U. Feige, S. Goldwasser, L. Lovász, S. Safra, and M. Szegedy. Approximating clique is almost np-complete. In *Proc. of FOCS*, 1991.

[16] T. A. Feo and M. G. Resende. Greedy randomized adaptive search procedures. *Journal of global optimization*, 6, 1995.

[17] N. Funabiki, Y. Takefuji, and K.-C. Lee. A neural network model for finding a near-maximum clique. *Journal of Parallel and Distributed Computing*, 14, 1992.

[18] D. Gibson, R. Kumar, and A. Tomkins. Discovering large dense subgraphs in massive graphs. In *Proc. of VLDB*, 2005.

[19] F. Glover and M. Laguna. *Tabu Search*. 2013.

[20] J. Håstad. Clique is hard to approximate within $n^{1\text{-epsilon}}$. In *Proc. of FOCS*, 1996.

[21] M. Jerrum. Large cliques elude the metropolis process. *Random Structures & Algorithms*, 3, 1992.

[22] D. S. Johnson. Approximation algorithms for combinatorial problems. *Journal of computer and system sciences*, 9, 1974.

[23] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*. 1972.

[24] I. Koch. Enumerating all connected maximal common subgraphs in two graphs. *Theoretical Computer Science*, 250(1-2):1–30, 2001.

[25] J. Konc and D. Janezic. An improved branch and bound algorithm for the maximum clique problem. *proteins*, 4(5), 2007.

[26] R. Kopf and G. Ruhe. A computational study of the weighted independent set problem for general graphs. *Found. Control Eng.*, 1987.

[27] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *Proc. of WWW'10*, 2010.

[28] T. Lappas, K. Liu, and E. Terzi. Finding a team of experts in social networks. In *Proc. of KDD*, 2009.

[29] J. Leskovec and R. Sosič. SNAP: A general purpose network analysis and graph mining library in C++. `http://snap.stanford.edu/snap`, 2014.

[30] K. Leung and C. Leckie. Unsupervised anomaly detection in network intrusion detection using clusters. In *Proc. of ACSW*, 2005.

[31] R. D. Luce and A. D. Perry. A method of matrix analysis of group structure. *Psychometrika*, 14, 1949.

[32] M. Mitzenmacher, J. Pachocki, R. Peng, C. Tsourakakis, and S. C. Xu. Scalable large near-clique detection in large-scale networks via sampling. In *Proc. of KDD*, 2015.

[33] M. Molloy and B. A. Reed. A critical point for random graphs with a given degree sequence. *Random structures and algorithms*, 1995.

[34] S. Niskanen and P. R. Östergård. *Cliquer User's Guide: Version 1.0*. Helsinki University of Technology Helsinki, Finland, 2003.

[35] P. R. Östergård. A fast algorithm for the maximum clique problem. *Discrete Applied Mathematics*, 120(1):197–207, 2002.

[36] S. Papadopoulos, Y. Kompatsiaris, A. Vakali, and P. Spyridonos. Community detection in social media. *Data Mining and Knowledge Discovery*, 24, 2012.

[37] B. Pattabiraman, M. M. A. Patwary, A. H. Gebremedhin, W.-k. Liao, and A. Choudhary. Fast algorithms for the maximum clique problem on massive sparse graphs. In *Prof. of WAW*, 2013.

[38] P. Prosser. Exact algorithms for maximum clique: A computational study. *Algorithms*, 5, 2012.

[39] W. Pullan and H. H. Hoos. Dynamic local search for the maximum clique problem. *Journal of Artificial Intelligence Research*, 25, 2006.

[40] R. A. Rossi, D. F. Gleich, A. H. Gebremedhin, and M. M. A. Patwary. Fast maximum clique algorithms for large graphs. In *WWW*, 2014.

[41] S. B. Seidman. Network structure and minimum degree. *Social networks*, 5, 1983.

[42] R. E. Tarjan and A. E. Trojanowski. Finding a maximum independent set. *SIAM Journal on Computing*, 6, 1977.

[43] E. Tomita and T. Kameda. An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments. *Journal of Global optimization*, 37, 2007.

[44] E. Tomita, Y. Sutani, T. Higashi, S. Takahashi, and M. Wakatsuki. A simple and faster branch-and-bound algorithm for finding a maximum clique. In *Proc. of WALCOM*, 2010.

[45] C. Tsourakakis, F. Bonchi, A. Gionis, F. Gullo, and M. Tsiarli. Denser than the densest subgraph: extracting optimal quasi-cliques with quality guarantees. In *Proc. of KDD*, 2013.

[46] J. Wang and J. Cheng. Truss decomposition in massive networks. *PVLDB*, 5, 2012.

[47] J. Wang, J. Cheng, and A. W.-C. Fu. Redundancy-aware maximal cliques. In *Proc. of SIGKDD*, pages 122–130, 2013.

[48] D. J. Watts and S. H. Strogatz. Collective dynamics of small-world networks. *Nature*, 393(6684), 1998.

[49] E. Yeger-Lotem, S. Sattath, N. Kashtan, S. Itzkovitz, R. Milo, R. Y. Pinter, U. Alon, and H. Margalit. Network motifs in integrated cellular networks of transcription–regulation and protein–protein interaction. *PNAS*, 101, 2004.

[50] L. Yuan, L. Qin, X. Lin, L. Chang, and W. Zhang. Diversified top-k clique search. In *Proc. of ICDE*, pages 387–398, 2015.

[51] Z. Zeng, J. Wang, L. Zhou, and G. Karypis. Coherent closed quasi-clique discovery from large dense graph databases. In *Proc. of KDD*, 2006.

[52] X. Zheng, T. Liu, Z. Yang, and J. Wang. Large cliques in arabidopsis gene coexpression network and motif discovery. *Journal of plant physiology*, 168, 2011.