Efficient Maximum Clique Computation over Large Sparse Graphs

Lijun Chang
The University of Sydney
Lijun.Chang@sydney.edu.au

ABSTRACT

This paper studies the problem of MCC-Sparse, Maximum Clique Computation over large real-world graphs that are usually *Sparse*. In the literature, MCC-Sparse has been studied separately and less extensively than its dense counterpart MCC-Dense, and advanced algorithmic techniques that are developed for MCC-Dense have not been utilized in the existing MCC-Sparse solvers. In this paper, we design an algorithm MC-BRB which transforms an instance of MCC-Sparse to instances of k-clique finding over dense subgraphs (KCF-Dense) that can be computed by the existing MCC-Dense solvers. To further improve the efficiency, we then develop a new *branch-reduce-&-bound* framework for KCF-Dense by proposing light-weight reducing techniques and leveraging the existing advanced branching and bounding techniques of MCC-Dense solvers. In addition, we also design an ego-centric algorithm MC-EGO for heuristically computing a near-maximum clique in near-linear time. We conduct extensive empirical studies on large real graphs and demonstrate the efficiency and effectiveness of our techniques.

CCS CONCEPTS

• **Mathematics of computing → Graph algorithms**; • **Information systems → Data mining**.

KEYWORDS

Maximum Clique; Large Sparse Graph; Branch-bound-and-reduce

ACM Reference Format:
Lijun Chang. 2019. Efficient Maximum Clique Computation over Large Sparse Graphs. In *The 25th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '19), August 4–8, 2019, Anchorage, AK, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3292500.3330986

1 INTRODUCTION

Graph model has been widely used to represent the relationships among entities in a wide spectrum of applications such as social networks, collaboration networks, communication networks and biological networks. As a result, we are nowadays facing a tremendous amount of large graphs that are *globally sparse but locally dense* [2], and it is of great importance to identify the locally dense subgraphs. In this paper, we study the problem of Maximum Clique Computation (MCC) over large sparse graphs, a fundamental problem in graph analytics. Formally speaking, a subset C of vertices in a graph G is a *clique* if there is an edge in G between every pair of vertices of C, and its *size* is measured by its number of vertices. A clique is *maximum* if its size is the largest among all cliques of G.

Applications. The MCC problem, besides being an important and interesting scientific problem itself, has many applications in graph-based data analytics [15, 17, 18]. For example, large cliques can be used as signals of rare or anomaly events such as terrorist recruitment [1] or web spam [12]. In bioinformatics and molecular biology, motif discovery requires to find large co-expression groups (i.e., cliques) in gene co-expression networks [25]. Finding large cliques can also accelerate other graph analysis tasks, e.g., subgraph enumeration which aims to enumerate all subgraphs of a large graph G that match an unlabeled query pattern q [10, 20]. It is easy to see that for a clique C of G, any subset with $|V(q)|$ vertices of C matches q regardless of q's structure.

On the other hand, computing the exact maximum clique can be used to evaluate the effectiveness of heuristic maximum clique computation algorithms, and the maximum clique size of G has an indication on the hardness of some analysis tasks on G. For example, as the maximum clique size of the graph uk-2002 is 944, subgraph enumeration over uk-2002 for any query with 6 vertices would have $\Omega(10^{18})$ results. This partially explains why the existing works only evaluate queries with up-to 5 vertices, *e.g.*, [20].

Existing Algorithms. The MCC problem is among Karp's original 21 NP-hard problems [9]. The existing algorithms are as follows.

(1) Heuristic Algorithms. It is shown in [8] that approximately computing a maximum clique within a factor of $n^{1-\epsilon}$ for any constant $0 < \epsilon < 1$ is NP-hard. As a result, heuristic techniques have been investigated for finding a large clique in practice [17, 18]. The general idea is to grow an initially empty clique C, by greedily moving vertices from a set R of candidate vertices to C. Here, R is maintained to be the set of vertices that are adjacent to all vertices of C, and is initialized by the vertices V of G. Maximum degree-based heuristic [17] and degeneracy order-based heuristic [18] greedily move to C the vertex of R that, respectively, has the maximum degree and has the highest rank according to the degeneracy ordering.

(2) Exact Algorithms. On the other hand, designing exact algorithms has also been extensively studied, *e.g.*, [3, 13, 14, 16–19, 21, 22, 24]. All these algorithms follow the *branch-&-bound* framework. It grows C and maintains R in a similar way to the above heuristic algorithms. However, it exhaustively tries moving each vertex of R to C, and generates a new *branch* (*i.e.*, a new instance C' and R') for each such movement. Upper *bounds* are computed for branches such that a branch is pruned if its upper bound is not better than the currently found largest clique. In the literature, various branching and bounding techniques have been developed [13, 14, 21, 22].

Motivation. The existing solutions for exact maximum clique computation (MCC) are divided into two groups:

- solvers for `MCC-Dense` (*i.e.*, MCC over Dense graphs) [3, 13, 14, 16, 21, 22, 24] that specifically handle dense graphs by *relying on the adjacency matrix graph representation*;
- solvers for `MCC-Sparse` (*i.e.*, MCC over Sparse graphs) [17–19] that handle large sparse graphs by *using the adjacency list graph representation.*

While `MCC-Dense` has been extensively and well studied, `MCC-Sparse` has been less-well studied. The existing `MCC-Dense` solvers cannot be directly applied to process large sparse graphs with millions of vertices due to the memory explosion of the adjacency matrix graph representation. However, advanced algorithmic techniques — such as graph recoloring [21, 22] and incremental MaxSAT reasoning [13, 14] — have been developed for `MCC-Dense`, whereas the existing `MCC-Sparse` solvers use only simple algorithmic techniques but resort to advanced programming techniques such as multi-core [18] and bit-parallel [19]. Two natural questions arise:

Question-I: is it possible to extend the existing `MCC-Dense` solvers to solve the problem of `MCC-Sparse`?
Question-II: if yes, then is the direct extension efficient?

Our Contributions. In this paper, we answer the above questions by building a bridge between `MCC-Sparse` and `MCC-Dense`. Our main contributions are as follows. We omit the proofs of all theorems and lemmas due to space limitation.

(1) Transform `MCC-Sparse` *to* `KCF-Dense` (Section 3). We design an exact algorithm MC-BRB for `MCC-Sparse` by transforming an instance of `MCC-Sparse` to instances of k-Clique Finding (`KCF`), each working on an ego-network. We prove that the maximum vertex number of the ego-networks is bounded by the *degeneracy* $\delta(G)$ with $\delta(G) \leq \lceil\sqrt{2m+n}\rceil$ for a graph G with n vertices and m edges. We show empirically that the ego-networks are dense. Thus, the input graphs to our `KCF` problem are small and dense. That is, we transform `MCC-Sparse` to `KCF-Dense` (*i.e.*, `KCF` over Dense graphs) which can be computed by the existing `MCC-Dense` solvers.

(2) A branch-reduce-&-bound Framework for `KCF-Dense` (Section 4). To further improve the efficiency, we develop a branch-reduce-&-bound framework for `KCF-Dense`, by introducing reducing techniques. Given an instance of `KCF-Dense` that consists of a dense graph g and an integer k, we propose reduction rules to reduce the size of g while preserving the existence of a k-clique. Specifically, applying the reduction rules outputs a graph g' and an integer k' such that $|V(g')| \leq |V(g)|$, $k' \leq k$, and g contains a k-clique if and only if g' contains a k'-clique. We also propose light-weight techniques to iteratively and exhaustively apply all our reduction rules to reduce g as much as possible, in amortized *linear time.*

(3) An Efficient Heuristic Algorithm MC-EGO (Section 5). To facilitate applications where finding a large clique suffices, we propose an efficient heuristic algorithm MC-EGO for finding a near-maximum clique in $O(\delta(G)\cdot m)$ time. This time complexity is near-linear, since $\delta(G)$ is theoretically bounded by $\lceil\sqrt{2m+n}\rceil$ and usually is at most thousands for real graphs as shown by our experiments.

(4) Extensive Empirical Studies (Section 6). We conduct extensive empirical studies on large real graphs to evaluate the efficiency and effectiveness of our algorithms. The results demonstrate that our exact algorithm MC-BRB outperforms the existing solutions of `MCC-Sparse` by up-to several orders of magnitude, and the clique reported by our heuristic algorithm MC-EGO either is certified to be or is very close to maximum. Most notably, for graphs `twitter-mpi` and `tech-p2p`, MC-BRB computes the maximum clique within five minutes while all existing algorithms take more than five hours.

Related Works. The related works are categorized as follows.

(1) Maximum Clique Computation. Besides the above mentioned heuristic algorithms and exact algorithms, a Monte Carlo algorithm RMC is recently proposed in [15]. Other techniques, such as parallelization [18] and distributed computing [24], are also investigated in the literature to speed up the computation. In this paper, we focus on single thread algorithms.

(2) Maximal Clique Enumeration. The problem of enumerating all maximal cliques in a graph has also been extensively studied (*e.g.*, see [7, 23]), with the state-of-the-art algorithm running in $O(d(n-d)3^{d/3})$ time where $d = \delta(G)$ is the degenerarcy of G. Although the largest one among all maximal cliques is the maximum clique, these algorithms are not efficient and not suitable for maximum clique computation due to lack of pruning techniques.

(3) Maximum Independent Set/Minimum Vertex Cover Computation. The problem of computing maximum independent set or equivalently computing minimum vertex cover is also studied in the literature (*e.g.*, see [4, 11]). Computing maximum independent set and computing maximum clique are theoretically equivalent; that is, $C \subseteq V$ is a maximum clique of G if and only if it is a maximum independent set of its complement graph $\overline{G} = (V, \overline{E})$, where $(u, v) \in \overline{E}$ if and only if $(u, v) \notin E$. Nevertheless, these techniques cannot be applied to our problem as the complement of a large sparse graph has an extremely large number of (*i.e.*, $\Omega(n^2)$) edges.

2 PRELIMINARIES

In this paper, we focus on *an unweighted undirected graph* $G = (V, E)$, where V is the set of vertices and E is the set of edges. We denote the number of vertices and the number of undirected edges in G by n and m, respectively. Let $(u, v) \in E$ denote an edge between u and v; u (resp. v) is said to be adjacent to and a neighbor of v (resp. u). The set of neighbors of u in G is $N_G(u) = \{v \in V \mid (u, v) \in E\}$, and the *degree* of u in G is $d_G(u) = |N_G(u)|$. Given a vertex subset S of G, we use $G[S]$ to denote the subgraph of G induced by S; that is, $G[S] = (S, \{(u, v) \in E \mid u, v \in S\})$. For ease of presentation, we simply refer to an unweighted undirected graph as a graph, and we abbreviate $N_G(u)$ and $d_G(u)$ as $N(u)$ and $d(u)$ when the graph in consideration is G. For an arbitrary given graph g, we denote the set of vertices and the set of edges of g by $V(g)$ and $E(g)$, respectively.

A vertex subset $C \subseteq V$ of a graph $G = (V, E)$ induces a *clique* if every pair of vertices of C is connected by an edge in G; we call the vertex set C a clique. The *size of a clique* C is measured by its number of vertices, denoted $|C|$. A clique C of G is a *maximal clique* if every proper superset of C in G is not a clique. A clique C of G is a *maximum clique* if its size is the largest among all (maximal) cliques of G, and this size is called the *clique number* of G, denoted $\omega(G)$. For example, consider the graph in Figure 1. $\{v_1, v_2, v_3\}$ is a clique of size 3, while $\{v_5, v_6, v_7, v_8\}$ is a maximum clique and the clique number of the graph is 4. Note that, the maximum clique may be not unique. For example, if we add an edge between v_3 and v_4, then $\{v_1, v_2, v_3, v_4\}$ is also a maximum clique besides $\{v_5, v_6, v_7, v_8\}$.

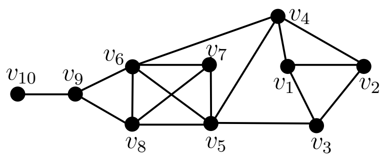


Figure 1: An example graph

Problem Statement. In this paper, we study the problem of maximum clique computation over a large sparse graph $G = (V, E)$, *i.e.*, MCC-Sparse, which aims to compute a maximum clique in G.

We consider a graph to be sparse if its density (*i.e.*, $\frac{2m}{n(n-1)}$) is below 0.1, which is the typical case for graphs with millions of vertices. For example, for the density of a 1-million-vertex graph to be over 0.1, the graph needs to have over 50 billion edges.

2.1 Core Number, Coloring, and Upper Bounds

We review three upper bounds for maximum clique computation.

Degree-based Upper Bound. The first upper bound is based on the degree information, as follows.

Lemma 2.1: *[18] For a graph G and a vertex u in G, the size of every clique in G containing u is no larger than $d(u) + 1$.*

It follows from Lemma 2.1 that $\omega(G) \leq \max_{u \in V} d(u) + 1$.

Core Number-based Upper Bound. The second upper bound is based on the concept of core number. Given a graph G, the ***core number*** of a vertex u, denoted $\texttt{core}(u)$, is the largest k such that u is in the k-core of G, where the k-core is the maximal subgraph whose minimum degree is at least k. The maximum core number of all vertices of G, denoted $\delta(G)$, is known as the ***degeneracy*** of G [7]. For the graph in Figure 1, the entire graph is a 1-core, the subgraph induced by vertices $\{v_1, \ldots, v_8\}$ is a 3-core, the core numbers of all vertices are shown in the second row of Table 1, and $\delta(G) = 3$. The core number-based upper bound is defined as follows.

Lemma 2.2: *[18] For a graph G and a vertex u in G, the size of every clique in G containing u is no larger than* $\texttt{core}(u) + 1$.

It follows from Lemma 2.2 that $\omega(G) \leq \max_{u \in V} \texttt{core}(u) + 1 = \delta(G) + 1$. As $\texttt{core}(u) \leq d(u)$ holds for every vertex $u \in V$, the core number-based upper bound is tighter than the degree-based upper bound. Note that, the core number for all vertices in G can be efficiently computed, *e.g.*, in $O(m)$ total time [5].

Graph Coloring-based Upper Bound. The third upper bound is based on the concept of graph coloring. Given a graph $G = (V, E)$, a *coloring* of G is to assign a ***color number***, denoted $\texttt{color}(u)$, to each vertex $u \in V$ such that no two adjacent vertices have the same color (*i.e.*, $\texttt{color}(u) \neq \texttt{color}(v), \forall (u, v) \in E$). For example, the last row of Table 1 shows a feasible coloring of the graph in Figure 1.

Lemma 2.3: *[21] For a graph G and a subset S of vertices of G, the maximum clique size of the subgraph $G[S]$ is no larger than the number of distinct colors of S, denoted* $\texttt{UniqColors}(S, \texttt{color}(\cdot))$.

It follows from Lemma 2.3 that $\omega(G) \leq \texttt{UniqColors}(V, \texttt{color}(\cdot))$. Thus, the fewer the number of distinct colors used in a graph coloring, the tighter the upper bound estimation. However, it is NP-hard to compute a graph coloring with the minimum number of distinct colors [9]. Thus, heuristics are usually adopted to color the vertices of a graph. One of the most widely used heuristics [21] is a greedy approach based on the degeneracy ordering of the vertices.

Table 1: Core numbers and a graph coloring

vertex	v_{10}	v_9	v_8	v_7	v_6	v_5	v_4	v_3	v_2	v_1
$\texttt{core}(\cdot)$	1	2	3	3	3	3	3	3	3	3
$\texttt{color}(\cdot)$	1	0	3	2	1	0	2	2	1	0

Definition 2.1:[5] A permutation $(v_1, v_2, \ldots, v_n)$ of all vertices of G is a ***degeneracy ordering*** if every vertex v_i has the smallest degree in the subgraph of G induced by $\{v_i, v_{i+1}, \ldots, v_n\}$.

Given a degeneracy ordering of all vertices of G, colors are greedily assigned to vertices in the reverse order: each vertex is assigned the smallest color that none of its neighbors have taken. It is immediate that the number of distinct colors is upper bounded by $\delta(G) + 1$. Thus, the coloring-based upper bound is tighter than the core number-based upper bound. For example, for the graph in Figure 1, the degeneracy ordering of the vertices is shown in the first row of Table 1, and the graph coloring based on this degeneracy ordering is shown in the last row. As a degeneracy ordering of the vertices in G can be computed in $O(m)$ time [5], the degeneracy ordering-based graph coloring can be conducted in $O(m)$ time.

3 TRANSFORM MCC-SPARSE TO KCF-DENSE

In this section, we propose techniques to transform the problem of MCC-Sparse to the problem of k-clique finding (KCF). The main idea is based on the following two lemmas; the proofs of all theorems and lemmas are omitted from the paper due to space limitation.

Lemma 3.1: *Given an arbitrary total ordering of V, we have $\omega(G) = \max_{u \in V} \omega\big(G[N^+(u)]\big) + 1$, where $G[N^+(u)]$ denotes the subgraph of G induced by the set $N^+(u)$ of higher-ranked neighbors of u according to the total ordering.*

Lemma 3.2: *Given an arbitrary total ordering $(v_1, v_2, \ldots, v_n)$ of V, we have $\omega\big(G[N^+(v_i)]\big) \leq \max_{j=i+1}^{n} \omega\big(G[N^+(v_j)]\big) + 1, \forall 1 \leq i < n$.*

We call $G[N^+(v_i)]$ an ***ego-network*** of G regarding vertex v_i; note that, all vertices of $G[N^+(v_i)]$ are connected to v_i. Figure 2 shows the ego-networks $G[N^+(v_3)]$ and $G[N^+(v_8)]$ for the graph in Figure 1 and the vertex ID-based total ordering $(v_{10}, v_9, \cdots, v_2, v_1)$.

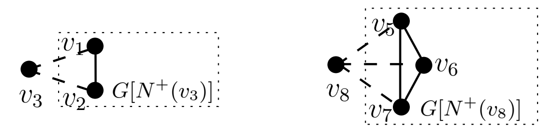


Figure 2: Ego-networks $G[N^+(v_3)]$ and $G[N^+(v_8)]$

Following Lemmas 3.1 and 3.2, we propose to process vertices in the reverse order according to a total ordering $(v_1, v_2, \ldots, v_n)$ of vertices of G. When processing vertex v_i, what we need to do is to find a clique of size k (aka, k-***clique***), if one exists, in the ego-network $G[N^+(v_i)]$; here, $k = \max_{j=i+1}^{n} \omega\big(G[N^+(v_j)]\big) + 1$ is the size of the currently found largest clique. If $G[N^+(v_i)]$ contains a k-clique C, then $\{v_i\} \cup C$ is a $(k + 1)$-clique of G and it updates the currently found largest clique. Finally, the stored largest clique is a maximum clique of G when the algorithm terminates. Thus, *our main problem for computing a maximum clique over a large sparse graph G becomes k-**clique finding** (KCF) in the n ego-networks of G.* For example, when processing the ego-network $G[N^+(v_8)]$, we should have found the 3-clique $\{v_1, v_2, v_3\}$, thus we will aim to find a 3-clique in $G[N^+(v_8)]$. As $\{v_5, v_6, v_7\}$ is a 3-clique in $G[N^+(v_8)]$, we obtain the 4-clique $\{v_5, v_6, v_7, v_8\}$.

Make Ego-networks Small and Dense. The above general idea applies to any total ordering of V. However, the number of vertices in the ego-networks $G[N^+(v_i)]$ can dramatically differ for different total orderings of V. We prove by the following two lemmas that the degeneracy ordering (see Definition 2.1) minimizes the maximum number of vertices in the n ego-networks; given a degeneracy ordering, we say a vertex ranks higher if it appears later in the degeneracy ordering.

Lemma 3.3: *If we use the degeneracy ordering of V, then we have $\max_{u \in V} |N^+(u)| \leq \delta(G) \leq \min\{d_{max}, \lceil\sqrt{2m+n}\rceil\}$, where d_{max} is the maximum vertex degree of G.*

Lemma 3.4: *For any total ordering of V, there is a vertex $u \in V$ such that $|N^+(u)| \geq \delta(G)$.*

Thus, we use the degeneracy ordering of vertices to construct the ego-networks in this paper. Our empirical studies (see Section 6) show that the ego-networks constructed for real graphs are dense. As a result, *the input graphs to our* KCF *problem are small and dense, and we have transformed* MCC-Sparse *to* KCF-Dense.

The MC-BRB **Algorithm.** Based on the above discussions, we propose an exact algorithm MC-BRB for MCC-Sparse. The pseudocode is shown in Algorithm 1. We first compute a heuristic initial clique C^*, a degeneracy ordering Dorder(·), a coloring color(·), and core numbers core(·) of G by invoking MC-EGO (Line 1), which will be discussed in Section 5. In addition, MC-EGO also returns an upper bound $\overline{\omega}$ of the clique number $\omega(G)$ such that Algorithm 1 can terminate at Line 2 if $|C^*| = \overline{\omega}$, *i.e.*, the heuristically computed clique is certified to be maximum. Then, we iteratively process the ego-networks $G[N^+(u)]$, for vertices $u \in V$ in reverse order (Lines 4–14), to find a $|C^*|$-clique in $G[N^+(u)]$ (Line 13) where C^* stores the currently found largest clique. If $G[N^+(u)]$ contains a $|C^*|$-clique C, then $\{u\} \cup C$ updates C^* (Line 14). As the ego-networks are small and dense, we represent them by an adjacency matrix (Line 12). Thus, we can invoke one of the existing MCC-Dense solvers (*e.g.*, [13, 14, 21, 22]) to compute KCF-Dense at Line 13.

To reduce the number of KCF-Dense instances to be generated at Line 13, we use core number-based upper bound (Line 5) and coloring-based upper bound (Line 7) to prune an ego-network if an upper bound of its clique number is smaller than the size of the currently found largest clique C^*. To facilitate efficient ego-network extraction, we orient the input graph G to obtain a directed graph G^+ based on the degeneracy ordering (Line 3), where each edge points from the lower-ranked end-point to the other end-point.

Analysis of MC-BRB**.** Due to the NP-hardness of MCC, the time complexity of Algorithm 1 is exponential to $\delta(G)$ in the worst-case. Recall that, Algorithm 1 generates n KCF-Dense instances in the worst-case, and the largest KCF-Dense instance contains $\delta(G)$ vertices. Our empirical studies in Section 6 show that the number of generated KCF-Dense instances is very small as a result of the pruning techniques at Lines 5–8.

It is worth mentioning that it is feasible to represent the ego-networks by an adjacency matrix at Line 12. Specifically, the size of

Algorithm 1: MC-BRB($G = (V, E)$)

```
1  ($C^*$, $\overline{\omega}$, Dorder(·), color(·), core(·)) ← MC-EGO($G$);
2  if $|C^*| < \overline{\omega}$ then
3      $G^+$ ← a directed graph by orienting $G$ w.r.t. Dorder(·);
4      for each vertex $u$ in reverse order w.r.t. Dorder(·) do
5          if core($u$) < $|C^*|$ then break;
6          $N^+(u)$ ← the set of $u$'s out-neighbors in $G^+$;
7          if UniqColors($N^+(u)$, color(·)) < $|C^*|$ then
8              continue;
9          Extract the subgraph $g$ of $G$ induced by $N^+(u)$;
10         Reduce $g$ to its ($|C^*| - 1$)-core;
11         if $g \neq \emptyset$ then
12             Construct an adjacency matrix A for $g$;
13             $C$ ← KCF-BRB(A, $V(g)$, $|C^*|$);   /* KCF-Dense */;
14             if $|C| = |C^*|$ then $C^*$ ← $\{u\} \cup C$;
15 return $C^*$;
```

the adjacency matrix is bounded by $(\delta(G))^2 \leq 2m+3n$ by following Lemma 3.3; note that, this bound is pessimistic, and in practice $\delta(G)$ is usually much smaller than $\sqrt{2m+n}$ for real graphs (*e.g.*, see Table 2 in Section 6). Moreover, to control the memory footprint, we represent the adjacency matrix **A** by a one-dimensional boolean array, which is simulated by a 32-bit integer array; that is, each bit represents one boolean value. Then, we have the following lemma.

Lemma 3.5: *The largest adjacency matrix constructed for $G[N^+(u)]$ for $u \in V$ can be represented by $\frac{(\delta(G))^2}{32} + \delta(G) \leq \frac{2m+n+34\sqrt{2m+n}}{32} + 1$ integers.*

As $m \leq \frac{n(n-1)}{2}$ holds in the worst case, we have $2m + n \leq n^2$. Thus, the largest adjacency matrix **A** can be represented by $\frac{2m+n+34n}{32} + 1 = \frac{m}{16} + \frac{35n}{32} + 1$ integers, which is small.

4 A BRANCH-REDUCE-&-BOUND FRAMEWORK FOR KCF-DENSE

Although the KCF-Dense instances generated in Algorithm 1 can be directly computed by the existing MCC-Dense solvers, this is inefficient as the existing MCC-Dense solvers are mainly optimized for (hard) graphs that typically have only thousands of vertices but take tens or hundreds of seconds to process. For example, in [14] the graphs that can be solved within 0.1 second are excluded from the experiments. However, the average processing time of our generated KCF-Dense instances is less than 0.1 seconds (see Section 6). Moreover, all existing MCC-Dense solvers follow the branch-&-bound framework. Motivated by this, we develop a new *branch-reduce-&-bound* framework for the problem of KCF-Dense in this section.

The pseudocode of our branch-reduce-&-bound framework, denoted KCF-BRB, is shown in Algorithm 2. We directly use the adjacency matrix **A** of a graph to refer the graph itself. Note that, we only store one copy of the adjacency matrix **A**, and a subgraph of **A** is represented by its vertex set R. Given **A**, a subset R of candidate vertices of **A**, and an integer $k > 0$, KCF-BRB returns a k-clique of $\mathbf{A}[R]$ (the subgraph of **A** induced by R) if there is one (Lines 3,9), and returns $\emptyset$ otherwise (Lines 5,11). To do so, we first *reduce* the problem instance by invoking the procedure Reduce which returns a clique C', and a smaller R' and k' (Line 2). Then,

Algorithm 2: KCF-BRB(A, R, k)

```
1  Obtain the degree $d(\cdot)$ of all vertices of $R$ in $\mathbf{A}[R]$;
2  $(C', R', k') \leftarrow$ Reduce(A, $R$, $k$, $d(\cdot)$);          /* Reduce */;
3  if $k' = 0$ then return $C'$;
4  $\texttt{color}_{R'}(\cdot) \leftarrow$ ReColoring(A, $R'$, $k' - 1$, $d(\cdot)$);     /* Bound */;
5  if UniqColors($R'$, $\texttt{color}_{R'}(\cdot)$) $< k'$ then return $\emptyset$;
6  Order $R'$ such that vertices with $\texttt{color}_{R'}(\cdot) \geq k' - 1$ are put at the
   front obeying the degeneracy ordering;
7  for each vertex $u$ in $R'$ s.t. $\texttt{color}_{R'}(u) \geq k' - 1$ do
8      $C \leftarrow$ KCF-BRB(A, $N_{R'}(u)$, $k' - 1$);          /* Branch */;
9      if $|C| = k' - 1$ then return $C \cup \{u\} \cup C'$;
10     $R' \leftarrow R' \backslash \{u\}$;
11 return $\emptyset$;
```

we compute an upper *bound* $\overline{\omega}$ of the clique number $\omega(\mathbf{A}[R'])$ of $\mathbf{A}[R']$ (Line 4), and return $\emptyset$ indicating that there is no k-clique in $\mathbf{A}[R]$ if $\overline{\omega}$ is smaller than k' (Line 5). If the problem instance is not pruned by the bounding technique, then we *branch* on each vertex $u \in R'$ (Line 7), according to a total ordering of R' (Line 6), to find a $(k'-1)$-clique of $\mathbf{A}[N_{R'}(u)]$ (Line 8), where $N_{R'}(u)$ denotes the set of neighbors of u in R'. When branching on vertex u, if a $(k'-1)$-clique C is found in $\mathbf{A}[N_{R'}(u)]$, then $C \cup \{u\} \cup C'$ is a k-clique of $\mathbf{A}[R]$ (Line 9); otherwise, u is removed from R'(Line 10).

The framework in Algorithm 2 is general, as different branching, reducing, and bounding techniques can be employed. In this paper, we propose new reducing techniques in Section 4.1, while adopting the branching and bounding techniques that have been used in the existing MCC-Dense solvers. For bounding, we use the recoloring technique proposed in [22] to reduce the number of distinct colors that are used; please refer to [22] for the details of recoloring. For branching, we use the light-weight technique in [14]. Specifically, given a degeneracy ordering and a coloring of R' and to compute a k'-clique in $\mathbf{A}[R']$, we reorder the vertices in R' such that all vertices with colors no smaller than $k'-1$ are put at the beginning by obeying the degeneracy ordering, while other vertices are put at the end (Line 6). Then, for each vertex u in R' with color at least $k'-1$ and according to this order, we generate a branch by including u into the clique (Line 8) and also a branch by excluding u (Line 10). Recall that, colors are integers starting from 0. Thus, every k'-clique of $\mathbf{A}[R']$ must contain a vertex with color no smaller than $k'-1$.

4.1 Our Reducing Technique

Given an instance of KCF-Dense (*i.e.*, a graph g and an integer k), we propose techniques to *safely remove* vertices from g and decrease k. Before that, we first introduce several notations. Given a vertex subset S of g, we use $g - S$ to denote the result of removing from g all vertices of S and their associated edges. Given vertices u_1 and u_2 of g, we use $g - \{u_1, u_2\} + u_{1,2}$ to denote the resulting graph by ***contracting*** u_1 and u_2 into a super-vertex, denoted $u_{1,2}$, while adjacent edges of u_1 and u_2 are preserved for $u_{1,2}$ based on the and-semantics. That is, $g - \{u_1, u_2\} + u_{1,2}$ is obtained from $g - \{u_1, u_2\}$ by adding a vertex $u_{1,2}$ and adding an edge between $u_{1,2}$ and every vertex of $g - \{u_1, u_2\}$ that are neighbors of *both* u_1 and u_2.

Degree-based Reduction Rules. We focus our discussions on a vertex u of g, and modify g and k correspondingly if $d_g(u) < k - 1$ or $d_g(u) \geq |V(g)| - 4$.

① *Low Degree Reduction Rule:* $d_g(u) < k - 1$. It is easy to see that if the degree $d_g(u)$ of u is less than $k-1$, then every clique in g that contains u must be of size smaller than k. Thus, we can safely remove u from g.

② *All Connection Reduction Rule:* $d_g(u) = |V(g)| - 1$. If u is adjacent to all other vertices of g, then every maximum clique in g includes u. Thus, we can remove u from g, and decrease k by 1.

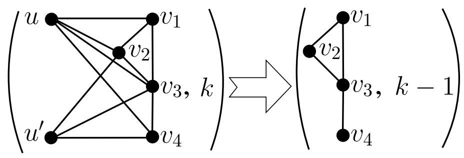
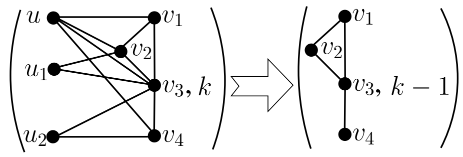
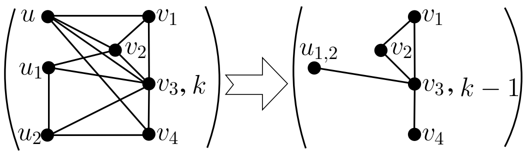


Figure 3: One neighbor missing reduction rule

③ *One Neighbor Missing Reduction Rule:* $d_g(u) = |V(g)| - 2$. If u is adjacent to all but one other vertices of g and let u' be the unique vertex that is not adjacent to u, then there exists a maximum clique in g that contains u (and thus does not contain u'). The reason is as follows: consider any maximum clique C in g that does not contain u (C must contain u', otherwise C is not maximal), then $(C \backslash \{u'\}) \cup \{u\}$ is a maximum clique that contains u but not u'. Thus, we can remove $\{u, u'\}$ from g, and decrease k by 1, see Figure 3.



(a) Case 1 of two neighbors missing reduction rule



(b) Case 2 of two neighbors missing reduction rule

Figure 4: Two neighbors missing reduction rules

④ *Two Neighbors Missing Reduction Rules:* $d_g(u) = |V(g)| - 3$. If u is adjacent to all but two other vertices of g and let u_1 and u_2 be the two non-adjacent vertices of u, then there are two cases depending on whether (u_1, u_2) is in $E(g)$.

Lemma 4.1: *Consider a vertex u in g that is adjacent to all vertices but u_1 and u_2 of g (see Figure 4).*

(1) $(u_1, u_2) \notin E(g)$*: then there exists a maximum clique in g that contains u (and thus contains none of $\{u_1, u_2\}$),* i.e., $\omega(g) = \omega(g - \{u, u_1, u_2\}) + 1$.

(2) $(u_1, u_2) \in E(g)$*: then there exists a maximum clique in g that contains either $\{u\}$ or $\{u_1, u_2\}$. Thus, let g' be the graph obtained from g by removing u and contracting u_1 and u_2* (i.e., $g - \{u, u_1, u_2\} + u_{1,2}$)*, we have* $\omega(g) = \omega(g') + 1$.

Following Lemma 4.1, the reduction rules for two neighbors missing are as follows. If $(u_1, u_2) \notin E(g)$, then we can safely remove $\{u, u_1, u_2\}$ from g and decrease k by 1, see Figure 4(a). Otherwise, we change g to $g - \{u, u_1, u_2\} + u_{1,2}$ (*i.e.*, remove u and contract u_1 and u_2) and decrease k by 1, see Figure 4(b).

⑤ *Three Neighbors Missing Reduction Rules:* $d_g(u) = |V(g)| - 4$. If u is adjacent to all but three other vertices of g and let u_1, u_2 and u_3 be the three non-adjacent vertices of u, then there are four cases depending on the number of edges in the subgraph of g induced by $\{u_1, u_2, u_3\}$; denote this set of edges by $E(\{u_1, u_2, u_3\})$.

Lemma 4.2: *Consider a vertex u in g that is adjacent to all vertices but u_1, u_2 and u_3 of g (see Figure 7 in Appendix).*

(1) $|E(\{u_1, u_2, u_3\})| = 0$*: then there exists a maximum clique in g that contains u (and thus contains none of $\{u_1, u_2, u_3\}$).*
(2) $|E(\{u_1, u_2, u_3\})| = 1$*: without loss of generality (w.l.o.g.) assume that the edge is between u_1 and u_2, then there exists a maximum clique in g that does not contain u_3.*
(3) $|E(\{u_1, u_2, u_3\})| = 2$*: w.l.o.g. assume that $(u_1, u_2) \in E(g)$ and $(u_2, u_3) \in E(g)$, let g' be the graph obtained from g by removing u, making a copy u_2' for u_2, and contracting $\{u_1, u_2\}$ and contracting $\{u_2', u_3\}$ (i.e., $g' = g - \{u, u_1, u_2, u_3\} + \{u_{1,2}, u_{2,3}\}$), then $\omega(g) = \omega(g') + 1$.*
(4) $|E(\{u_1, u_2, u_3\})| = 3$*: let g'' be the graph obtained from g by removing u, making a copy u_1', u_2', and u_3', respectively, for u_1, u_2, and u_3, and contracting $\{u_1, u_2'\}$, contracting $\{u_2, u_3'\}$ and contracting $\{u_3, u_1'\}$, and adding an edge between $u_{1,2}$ and $u_{2,3}$ (i.e., $g'' = g - \{u, u_1, u_2, u_3\} + \{u_{1,2}, u_{2,3}, u_{3,1}\} + \{(u_{1,2}, u_{2,3})\}$), then $\omega(g) = \omega(g'') + 1$.*

Please see the four subfigures in Figure 7 in Appendix for an example of the above four cases. Similar to the reduction rules for two neighbors missing, it is straightforward to design reduction rules correspondingly for the four cases of three neighbors missing by following Lemma 4.2; we omit the details. The number of vertices of g is reduced by 4, 3, 2 and 1, respectively, for the four cases.

Efficiently Apply The Reduction Rules. Intuitively, the more reduction rules applied, the smaller the resulting graph. Thus, we propose to iteratively and exhaustively apply all the reduction rules one after another until no reduction rule can be applied to the resulting graph. The pseudocode of iteratively and exhaustively applying all the degree-based reduction rules, denoted Reduce, is shown in Algorithm 3, which runs in iterations. In each iteration, it sequentially checks each vertex u of R (Line 4), and applies the corresponding reduction rule if $d(u) < k - 1$ or $d(u) \geq |R| - 4$ (Lines 5–6). As each reduction rule is correct, Algorithm 3 is correct regardless of the ordering of applying the reduction rules. Note that the reduction rules are applied to the *dense* ego-networks of the input graph G, rather than directly to the *sparse* input graph G.

There are two things worth mentioning regarding the implementation. Firstly, the last two reduction rules may modify the adjacency matrix **A** due to contracting vertices (see case 2 of Lemma 4.1 and cases 3–4 of Lemma 4.2). In our implementation, to contract vertices u_1 and u_2, we use u_1 to represent the super-vertex $u_{1,2}$ by removing the edge between u_1 and each of its neighbors that is not adjacent to u_2. Thus, the only modification to **A** is removing edges. Note that, all modifications to **A** are restored in Algorithm 2 when backtracking. Secondly, when applying two neighbors missing and three neighbors missing reduction rules, we are sure that one more vertex will be added to C after processing the resulting graph, however we do not know at this moment which additional vertex will be added (see the proofs of Lemma 4.1 and Lemma 4.2). Thus, we add a dummy vertex into C and also bookkeep auxiliary information for this situation, and then replace the dummy vertex with a proper vertex based on the stored auxiliary information in a post-processing phase. We omit the details from Algorithm 2 and Algorithm 3 for presentation simplicity.

Algorithm 3: Reduce($\mathbf{A}, R, k, d(\cdot)$)

```
1 $C \leftarrow \emptyset$;
2 while true do
3     $preSize \leftarrow |R|$;
4     for each vertex $u$ in $R$ do
5         if $d(u) < k - 1$ or $d(u) \geq |R| - 4$ then
6             Apply the corresponding reduction rule for $u$;
7     if $|R| = preSize$ then break;
8 return ($C$, $R$, $k$);
```

Lemma 4.3: *The time complexity of Algorithm 3 is $O((|R| - |R'| + 1) \cdot |R|)$, where R is the input set of candidate vertices and R' is the output set of candidate vertices.*

The time complexity of Algorithm 3 is proved by Lemma 4.3 in above. Note that, Algorithm 3 is light-weight: (1) the amortized time complexity of reducing one vertex is $O(|R|)$, and (2) for graphs that Algorithm 3 cannot reduce any vertex, the running time is bounded by $O(|R|)$ which is negligible.

5 OUR HEURISTIC ALGORITHM MC-EGO

In this section, we propose a heuristic algorithm MC-EGO for computing a near-maximum clique in near-linear time. As the maximum clique in many real world graphs can be easily found (see the first evaluation in Section 6.1), we first present a linear time algorithm MC-DD. In a nutshell, MC-DD computes a maximum degree-based clique and a degeneracy-based clique, and returns the one with a larger cardinality. The ***degeneracy-based clique*** is obtained as the longest suffix of the degeneracy ordering that is a clique. The pseudocode of MC-DD is shown and discussed in Section A.2 in Appendix. Note that, besides a clique C^*, MC-DD also outputs an upper bound $\overline{\omega}$ of the clique number of G, such that C^* is certified to be a maximum clique if $|C^*| = \overline{\omega}$. MC-DD runs in $O(m)$ time as the degeneracy ordering can be computed in $O(m)$ time [5].

Our heuristic algorithm MC-EGO invokes MC-DD such that it stops in linear time if MC-DD computes and certifies a maximum clique. Otherwise, MC-EGO computes a degeneracy-based clique for every ego-network $G[N^+(u)]$ of G, and report the largest one. Thus, MC-EGO is an ego-centric degeneracy-based greedy algorithm. The pseudocode of MC-EGO is shown in Algorithm 4, which follows the same structure as the exact algorithm MC-BRB (Algorithm 1) with two differences. Firstly, rather than invoking KCF-BRB for finding a k-clique in an ego-network $G[N^+(u)]$ which is NP-complete, we compute a degeneracy-based clique in $G[N^+(u)]$ in linear time to the size of $G[N^+(u)]$ (Lines 11-12). Secondly, we also compute a tighter upper bound of $\omega(G)$ which is obtained as the largest value among the clique number upper bounds computed for the ego-networks (Lines 14–16); the correctness of the tighter upper bound directly follows from Lemma 3.1.

The time complexity of MC-EGO is $O(\delta(G) \cdot m)$ as proved in the theorem below. This time complexity is near-linear as $\delta(G)$ is bounded by $\lceil \sqrt{2m + n} \rceil$ and is at most thousands for real graphs

Algorithm 4: MC-EGO($G = (V, E)$)

```
1  ($C^*$, $\overline{\omega}$, Dorder(·), color(·), core(·)) ← MC-DD($G$);
2  if $|C^*| < \overline{\omega}$ then
3      $\overline{\omega}' \leftarrow |C^*|$;
4      $G^+ \leftarrow$ a directed graph by orienting $G$ w.r.t. Dorder(·);
5      for each vertex $u$ in reverse order w.r.t. Dorder(·) do
6          The same as Lines 5–9 of Algorithm 1, and $g = G[N^+(u)]$;
7          $C \leftarrow$ compute the degeneracy-based clique of $g$;
8          if $|C| \geq |C^*|$ then $C^* \leftarrow \{u\} \cup C$;
9          Compute a graph coloring color′(·) for $g$;
10         if UniqColors($V(g)$, color′(·)) + 1 > $\overline{\omega}'$ then
11             $\overline{\omega}' \leftarrow$ UniqColors($V(g)$, color′(·)) + 1;
12     if $\overline{\omega}' < \overline{\omega}$ then $\overline{\omega} \leftarrow \overline{\omega}'$;
13 return ($C^*$, $\overline{\omega}$, Dorder(·), color(·), core(·));
```

(see Table 2 in Section 6). It is worth mentioning that for the graphs that MC-DD computes and certifies a maximum clique, MC-EGO also computes a maximum clique in linear time.

Theorem 5.1: MC-EGO *runs in* $O(\delta(G) \cdot m)$ *time.*

6 EXPERIMENTS

In this section, we evaluate the performance of our algorithms for computing maximum cliques in large real graphs. The main goals of this experimental study are as follows.

(1) We evaluate our exact algorithm MC-BRB against the existing algorithms for MCC-Sparse, and show that MC-BRB outperforms all existing algorithms. (Section 6.1)
(2) We evaluate the individual techniques used in MC-BRB, and show that each of the techniques contributed to the efficiency of MC-BRB. (Section 6.2)
(3) We evaluate our heuristic algorithm MC-EGO, and show that it returns near-maximum cliques (*i.e.*, with a gap of at most 3) for all the tested graphs. (Section 6.3)

Compared Algorithms. We evaluate the following exact algorithms for MCC-Sparse.

Existing Exact Algorithms. We include three state-of-the-art algorithms for MCC-Sparse: PMC [18], BBMCSP [19], and RMC [15]. Note that, none of the existing MCC-Dense solvers can process the graphs we tested.

Our Exact Algorithms. Besides MC-BRB [1], we also implemented several of its variants to evaluate the individual techniques used in MC-BRB (see the description in Section 6.2).

In addition, we also evaluate the two heuristic algorithms presented in Section 5: MC-DD and MC-EGO. All algorithms are implemented in C++, and run in a single-thread mode. The bit-parallel technique of BBMCSP is enabled in our testings.

Datasets. We evaluate the algorithms on 19 large real graphs from different domains, which are downloaded from the Stanford Network Analysis Platform [2], the Laboratory of Web Algorithmics [3] and the Network Repository [4]. Descriptions of the graphs can also be found there. Statistics of the graphs are shown in Table 5 in Appendix.

[1]The source code of MC-BRB is available at https://github.com/LijunChang/MC-BRB
[2]http://snap.stanford.edu/
[3]http://law.di.unimi.it/datasets.php
[4]http://networkrepository.com/

Measures. We measure the *running time* and *memory usage* of the algorithms. The reported running time is the total CPU time excluding only the I/O time of loading graph from disk to main memory, and a timeout of 5 hours is set. The reported memory usage is "the maximum resident set size of the process during its lifetime", as measured by the Linux command /usr/bin/time.[5] Experiments are conducted on a machine with an Intel(R) Xeon(R) 3.4GHz CPU and 16GB main memory running Linux (64bit Debian).

6.1 Against the Existing Algorithms

We evaluate the efficiency of our exact algorithm MC-BRB against the state-of-the-art algorithms: PMC, BBMCSP, and RMC. We also tested FMC, and the results show that it is consistently outperformed by PMC, BBMCSP and RMC, conforming to the existing experimental studies in [15, 18, 19]; thus, we omit FMC.

Evaluate the Hardness of Graph Instances. To show the hardness of a graph instance for maximum clique computation, we record the stage at which MC-BRB terminates. Recall that, MC-BRB invokes MC-EGO for heuristically computing an initial clique which in turn invokes MC-DD. Thus, we say MC-BRB terminates at stage "S1" if it terminates immediately after MC-DD, at stage "S2" if it terminates immediately after MC-EGO, and at stage "S3" otherwise. That is, a maximum clique is found and certified in linear time (*i.e.*, $O(m)$) by MC-DD and in near-linear time (*i.e.*, $O(\delta(G) \cdot m)$) by MC-EGO, respectively, if MC-BRB terminates at stages S1 and S2; for these two cases, we say the graph is an easy instance for MCC. From the last column of Table 2, we can see that among the 19 real graphs, the maximum clique is computed and certified by MC-DD for six graphs, and by MC-EGO for four graphs.

Evaluate Running Time Against Existing Algorithms. The running time of PMC, BBMCSP, RMC and MC-BRB are shown in the 5th, 8th, 11th, and 14th columns of Table 2, respectively. We can see that MC-BRB consistently runs faster than all existing algorithms. Firstly, it is worth noting that RMC is a Monte Carlo algorithm which may not always output a maximum clique; for example, on graphs human-gene2 and as-Skitter, it has an absolute error of 1. Secondly, the speedup of MC-BRB over the existing algorithms can be up-to several orders of magnitude. For example, for graphs human-gene1, twitter-mpi, and tech-p2p, MC-BRB computes a maximum clique under five minutes while none of the existing algorithms can finish within five hours. Note that, we also modified the maximal clique enumeration algorithm in [7] to compute maximum clique by pruning a branch if $|P| + |R|$ is no larger than the currently found largest clique. However, it does not perform well; for example, the running time on wiki-Talk and as-Skitter are 40 and 23 seconds, respectively.

Evaluate Memory Usage Against Existing Algorithms. We report the memory usage of these algorithms in the 6th, 9th, 12th and 15th columns of Table 2. The main objective is to show that using adjacency matrices to represent ego-networks in MC-BRB is feasible in terms of memory consumption. We can see that MC-BRB has a smaller memory footprint than the existing algorithms; this is mainly due to the compressed sparse row (CSR) representation of the input graph in MC-BRB. We further illustrate in Figure 5 the

[5]http://man7.org/linux/man-pages/man1/time.1.html

Table 2: Running time and memory usage of MC-BRB against existing algorithms (Stg: Stage); note that RMC has an additive error of 1 for `human-gene2` and `as-Skitter`

Graph	$\omega(G)$	$\delta(G)$	PMC			BBMCSP			RMC			MC-BRB			
			$\|C_0\|$	Time	Mem	$\|C_0\|$	Time	Mem	$\|C_0\|$	Time	Mem	$\|C_0\|$	Time	Mem	Stg
`Email`	16	37	15	0.07s	69M	13	0.2s	38M	16	0.064s	38M	16	**0.016s**	**15M**	S3
`Epinions`	23	67	22	0.2s	34M	20	0.18s	25M	23	0.12s	18M	23	**0.061s**	**10M**	S3
`slashdot`	26	54	24	0.13s	32M	22	0.216s	30M	24	0.065s	18M	26	**0.021s**	**10M**	S3
`DBLP`	114	113	114	0.047s	62M	114	0.086s	69M	114	0.055s	57M	114	**0.019s**	**20M**	S1
`Amazon`	11	10	11	0.095s	115M	11	0.171s	112M	11	0.1s	84M	11	**0.065s**	**38M**	S1
`Google`	44	44	44	0.368s	207M	44	2.06s	273M	44	0.282s	168M	44	**0.149s**	**72M**	S1
`wiki-Talk`	26	131	22	4.8s	843M	16	6.8s	385M	22	3.5s	389M	26	**0.75s**	**136M**	S3
`human-gene2`	1,300	1,902	1,250	5,996s	415M	-	>5h	-	1,240	*95s*	160M	1,300	**43s**	**94M**	S3
`as-Skitter`	67	111	66	1.21s	574M	50	5.54s	502M	66	*0.92s*	350M	67	**0.26s**	**157M**	S2
`human-gene1`	1,335	2,047	-	>5h	-	-	>5h	-	-	>5h	-	1,335	**76s**	**126M**	S3
`soc-flickr`	98	568	77	663s	857M	68	98.8s	698M	74	3,877s	786M	96	**40s**	**227M**	S3
`patent`	11	64	11	3.8s	844M	10	12.6s	1G	11	3.9s	691M	11	**2.5s**	**294M**	S2
`soc-pokec`	29	47	29	8.09s	810M	29	19s	1.1G	29	3.62s	469M	29	**3.09s**	**309M**	S2
`LiveJ`	321	372	302	4.68s	1.4G	314	23.15s	2G	321	3.94s	1.1G	321	**2.27s**	**532M**	S2
`twitter-mpi`	131	677	-	>5h	-	-	>5h	-	-	>5h	-	130	**239s**	**1.3G**	S3
`tech-p2p`	178	853	-	>5h	-	-	>5h	-	-	>5h	-	175	**263s**	**1.5G**	S3
`uk-2002`	944	943	944	7.2s	7.6G	944	9.4s	2.7G	944	7.8s	5.3G	944	**1.7s**	**2.6G**	S1
`webbase`	1,507	1,506	-	-	oom	1,507	52s	15G	-	-	oom	1,507	**7.8s**	**10G**	S1
`it-2004`	3,222	3,224	-	-	oom	-	-	oom	3,222	240s	15G	3,222	**8.4s**	**9G**	S1

Table 3: Running time of our algorithms (s: seconds, h: hours)

Graph	MC-EGO	MC-BRB	MC-BRB/I	MC-BRB/R	MC-BRB/C	MC-BRB/B	MC-A/IR	MC-M/IR	MC-MoMC
`Email`	0.016s	0.016s	0.021s	0.016s	0.016s	0.016s	0.023s	0.020s	0.016s
`Epinions`	0.061s	0.061s	0.060s	0.061s	0.061s	0.061s	0.080s	0.076s	0.062s
`slashdot`	0.021s	0.021s	0.022s	0.021s	0.021s	0.021s	0.022s	0.022s	0.022s
`wiki-Talk`	0.64s	0.75s	0.72s	0.72s	0.77s	0.75s	0.78s	0.74s	1.44s
`human-gene2`	31s	43s	35s	62s	43s	43s	150s	49s	>5h
`human-gene1`	53.8s	76s	61s	559s	77s	76s	1656s	528s	>5h
`soc-flickr`	14.8s	40s	50s	40s	61s	42s	174s	63s	1,485s
`twitter-mpi`	33s	239s	421s	304s	612s	251s	1,700s	536s	1,847s
`tech-p2p`	115s	263s	274s	634s	481s	277s	2,274s	615s	3,516s

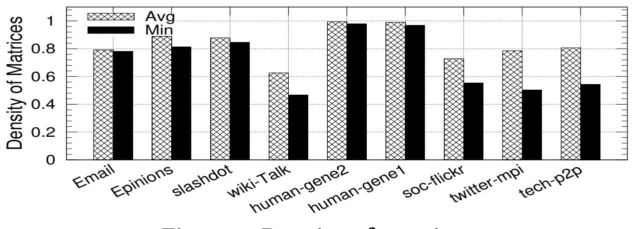


Figure 5: Density of matrices

average and the minimum density of the ego-networks that are represented by adjacency matrices ($\frac{2m}{n(n-1)}$). We can see that they are above 0.5 and 0.4, respectively, across all these graphs. Moreover, as shown in the third column of Table 2, the degeneracy $\delta(G)$, which bounds the maximum number of vertices in the ego-networks, is small for real graphs. Thus, the memory overhead of representing ego-networks by adjacency matrices is negligible.

6.2 Individual Techniques in MC-BRB

We now evaluate the effectiveness of the different techniques that are used in MC-BRB, by implementing the following variants.

- MC-MoMC: invoke the existing `MCC-Dense` solver MoMC [6] in [14] for computing `KCF-Dense` at Line 13 of Algorithm 1.
- MC-BRB/I: MC-BRB without initial clique computation.
- MC-BRB/R: MC-BRB without reducing.
- MC-BRB/C: MC-BRB without recoloring.
- MC-BRB/B: MC-BRB without advanced branching.

We run the algorithms on the graphs that MC-EGO *does not certify a maximum clique.* Table 3 shows the running time. Table 4 in Appendix illustrates the number of `KCF-Dense` instances (#`KCF-Dense`) generated at Line 13 of Algorithm 1, the total number of invocations of Algorithm 2 (#branches) which indicates the search space, and the maximum depth (D) of recursively invoking Algorithm 2.

Invoke Existing `MCC-Dense` Solvers for `KCF-Dense`. The running time of MC-MoMC is shown in the last column of Table 3. We can see that MC-MoMC, despite of having a large overhead incurred by MoMC, runs faster than the existing algorithms in Table 2. This demonstrates the superiority of our strategy by transforming the problem of `MCC-Sparse` to the problem of `KCF-Dense`. Nevertheless, from the second and third columns of Table 3 and the second column of Table 4, we can infer that the average processing time of the generated `KCF-Dense` instances is less than 0.1 seconds, which is out of the scope of the existing `MCC-Dense` solvers' optimization goals. This motivates us to propose a new algorithm KCF-BRB for the `KCF-Dense` instances generated from `MCC-Sparse`.

Evaluate The Effect of Initial Clique Computation. By comparing the results of MC-BRB/I with MC-BRB in Tables 3 and 4, we can see that computing a larger initial clique in MC-BRB, although reduces the number of `KCF-Dense` instances, has a mixed effect

[6]http://www.mis.u-picardie.fr/~cli/MoMC2016.c

on the total running time. This is because, MC-BRB/I, although does not invoke MC-EGO nor MC-DD, inherently computes the degeneracy-based clique due to our strategy of processing the ego-networks in the reverse order according to the degeneracy ordering. Thus, if MC-EGO does not compute a much larger clique than the degeneracy-based clique, then MC-BRB can run slower than MC-BRB/I due to the overhead of MC-EGO (*e.g.*, on human-gene1 and human-gene2) as shown in the second column of Table 3. Nevertheless, MC-BRB is more robust than MC-BRB/I.

Evaluate Reducing, Bounding, and Branching. To evaluate the effect of reducing, bounding and branching, we evaluate MC-BRB against MC-BRB/R, MC-BRB/C and MC-BRB/B. Here, MC-BRB/R does not use reducing, MC-BRB/C uses the standard greedy coloring strategy without recoloring, and MC-BRB/B uses the highest color-based branching technique, as done in PMC and BBMCSP.

By comparing the running time and #Branches of the four algorithms, we can see that all the three techniques (*i.e.*, reducing, bounding and branching) have positive effects. Note that, all the four algorithms process the same number of KCF-Dense instances, and they differ only in how each KCF-Dense instance is solved. Firstly, we separate the running time of MC-BRB/R and MC-BRB into initial time (*i.e.*, the time of MC-EGO) and search time (*i.e.*, the total time minus the initial time). We can see that the reducing technique speeds up the search time by 22 and 3.5 times, respectively, for human-gene1 (from 505.2*s* to 22.2*s*) and tech-p2p (from 519*s* to 148*s*); this can be explained by the significantly reduced search space (*i.e.*, #Branches). Secondly, we can see that both the recoloring optimization and the branching technique that are adopted from MCC-Dense improve the running time.

Evaluate Adjacency Matrix Representation. To evaluate the impact of representing ego-networks by adjacency matrices, we implemented another two variants of MC-BRB that only differ from each other in the graph representation: MC-A/IR and MC-M/IR use adjacency array and adjacency matrix, respectively. The running time is shown in the 8–9th columns of Table 3. We can see that representing ego-networks by adjacency matrices improves the performance. This is due to the more efficient access of a vertex's neighbors in an induced subgraph based on an adjacency matrix.

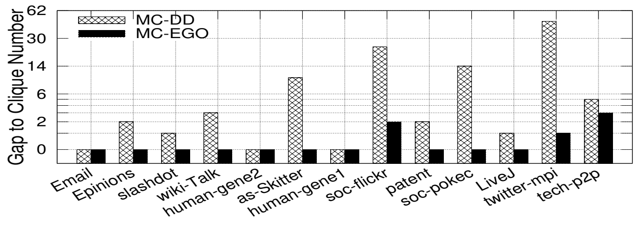


Figure 6: Gap to the clique number $\omega(G)$

6.3 Effectiveness of MC-EGO

Figure 6 shows the gap between the clique reported by MC-EGO (respectively, MC-DD) and the clique number $\omega(G)$, for the graphs that MC-DD does not certify a maximum clique (*i.e.*, $\overline{\omega} > |C^*|$). We can see that MC-EGO has a much smaller gap than MC-DD (*e.g.*, 1 v.s. 47 for twitter-mpi), and the gap for MC-EGO is at most 3. Thus, MC-EGO computes a near-maximum clique, and can be used in practice for efficiently computing a high-quality clique.

7 CONCLUSION

In this paper, we built a bridge between MCC over sparse graphs (MCC-Sparse) and MCC over dense graphs (MCC-Dense) by transforming an instance of MCC-Sparse to instances of KCF-Dense, and developed a branch-reduce-&-bound framework for KCF-Dense. We designed an exact algorithm MC-BRB, and an ego-centric heuristic algorithm MC-EGO. Experimental results on large real graphs demonstrated the efficiency and effectiveness of our algorithms.

Acknowledgements. The author is supported by ARC DP160101513 and FT180100256.

REFERENCES

[1] Nina Berry, Teresa Ko, Tim Moy, Julienne Smrcka, Jessica Turnley, and Ben Wu. 2004. Emergent Clique Formation in Terrorist Recruitmen. In *Workshop on Agent Organizations: Theory and Practice.*
[2] S. Boccaletti, V. Latora, Y. Moreno, M. Chavez, and D.-U. Hwang. 2006. Complex networks: Structure and dynamics. *Physics Reports* 424, 4-5 (2006), 175 – 308.
[3] Randy Carraghan and Panos M. Pardalos. 1990. An Exact Algorithm for the Maximum Clique Problem. *Oper. Res. Lett.* 9, 6 (Nov. 1990), 375–382.
[4] Lijun Chang, Wei Li, and Wenjie Zhang. 2017. Computing A Near-Maximum Independent Set in Linear Time by Reducing-Peeling. In *Proc. of SIGMOD'17.*
[5] Lijun Chang and Lu Qin. 2018. *Cohesive Subgraph Computation over Large Sparse Graphs.* Springer Series in the Data Sciences.
[6] Norishige Chiba and Takao Nishizeki. 1985. Arboricity and Subgraph Listing Algorithms. *SIAM J. Comput.* 14, 1 (1985), 210–223.
[7] David Eppstein, Maarten Löffler, and Darren Strash. 2013. Listing All Maximal Cliques in Large Sparse Real-World Graphs. *ACM Journal of Experimental Algorithmics* 18 (2013). https://doi.org/10.1145/2543629
[8] Johan Håstad. 1996. Clique is Hard to Approximate Within n$^{1\text{-epsilon}}$. In *Proc. of FOCS'96.* 627–636.
[9] Richard M. Karp. 1972. Reducibility Among Combinatorial Problems. In *Proc. of CCC'72.* 85–103.
[10] Hyeonji Kim, Juneyoung Lee, Sourav S. Bhowmick, Wook-Shin Han, Jeong-Hoon Lee, Seongyun Ko, and Moath H. A. Jarrah. 2016. DUALSIM: Parallel Subgraph Enumeration in a Massive Graph on a Single Machine. In *Proc. of SIGMOD'16.*
[11] Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Renato F. Werneck. 2016. Finding Near-Optimal Independent Sets at Scale. In *Proc. of ALENEX'16.* 138–150.
[12] Kingsly Leung and Christopher Leckie. 2005. Unsupervised Anomaly Detection in Network Intrusion Detection Using Clusters. In *Proc. of ACCS'05.* 333–342.
[13] Chu-Min Li, Zhiwen Fang, and Ke Xu. 2013. Combining MaxSAT Reasoning and Incremental Upper Bound for the Maximum Clique Problem. In *Proc. of ICTAI'13.*
[14] Chu-Min Li, Hua Jiang, and Felip Manyà. 2017. On minimization of the number of branches in branch-and-bound algorithms for the maximum clique problem. *Computers & OR* 84 (2017), 1–15.
[15] Can Lu, Jeffrey Xu Yu, Hao Wei, and Yikai Zhang. 2017. Finding the Maximum Clique in Massive Graphs. *PVLDB* 10, 11 (2017), 1538 – 1549.
[16] Panos M. Pardalos and Jue Xue. 1994. The maximum clique problem. *J. global Optimization* 4, 3 (1994), 301–328.
[17] Bharath Pattabiraman, Md. Mostofa Ali Patwary, Assefaw Hadish Gebremedhin, Wei-keng Liao, and Alok N. Choudhary. 2015. Fast Algorithms for the Maximum Clique Problem on Massive Graphs with Applications to Overlapping Community Detection. *Internet Mathematics* 11, 4-5 (2015), 421–448.
[18] Ryan A. Rossi, David F. Gleich, and Assefaw Hadish Gebremedhin. 2015. Parallel Maximum Clique Algorithms with Applications to Network Analysis. *SIAM J. Scientific Computing* 37, 5 (2015).
[19] Pablo San Segundo, Alvaro Lopez, and Panos M. Pardalos. 2016. A new exact maximum clique algorithm for large and massive sparse graphs. *Computers & Operations Research* 66 (2016), 81–94.
[20] Marco Serafini, Gianmarco De Francisci Morales, and Georgos Siganos. 2017. QFrag: distributed graph search via subgraph isomorphism. In *Proc. of SoCC'17.*
[21] Etsuji Tomita. 2017. Efficient Algorithms for Finding Maximum and Maximal Cliques and Their Applications. In *Proc. of WALCOM'17.* 3–15.
[22] Etsuji Tomita, Yoichi Sutani, Takanori Higashi, Shinya Takahashi, and Mitsuo Wakatsuki. 2010. A simple and faster branch-and-bound algorithm for finding a maximum clique. In *Proc. of WALCOM'10.* 191–203.
[23] Etsuji Tomita, Akira Tanaka, and Haruhisa Takahashi. 2006. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theor. Comput. Sci.* 363, 1 (2006), 28–42.
[24] Jingen Xiang, Cong Guo, and Ashraf Aboulnaga. 2013. Scalable maximum clique computation using mapreduce. In *Proc. of ICDE'13.* 74–85.
[25] Xiaoqi Zheng, Taigang Liu, Zhongnan Yang, and Jun Wang. 2011. Large cliques in Arabidopsis gene coexpression network and motif discovery. *Journal of plant physiology* 168, 6 (2011), 611–618.

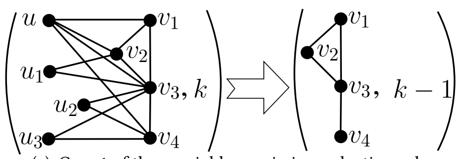
Table 4: Number of KCF-Dense instances (#KCF-Dense) and branches (#Branches), and search depth (D)

Graph	MC-BRB			MC-BRB/I			MC-BRB/R		MC-BRB/C		MC-BRB/B	
	#KCF-Dense	#Branches	D	#KCF-Dense	#Branches	D	#Branches	D	#Branches	D	#Branches	D
Email	2	2	1	83	83	1	2	1	2	1	2	1
Epinions	8	8	1	253	253	1	10	2	8	1	8	1
slashdot	4	4	1	108	108	1	4	1	4	1	4	1
wiki-Talk	485	1,934	3	1,447	2,896	3	2,114	3	3,465	3	1,934	3
human-gene2	653	653	1	1,541	1,541	1	3,308	9	653	1	653	1
human-gene1	928	991	2	2,110	2,173	2	43,691	11	1,491	3	991	2
soc-flickr	1,660	63,102	3	6,331	189,449	5	70,985	5	174,767	5	65,891	4
twitter-mpi	2,761	770,581	7	9,093	1,611,461	7	983,321	9	3,483,581	7	796,121	6
tech-p2p	2,044	471,320	6	12,105	481, 381	6	1,418,101	11	1,616,649	7	516,135	6

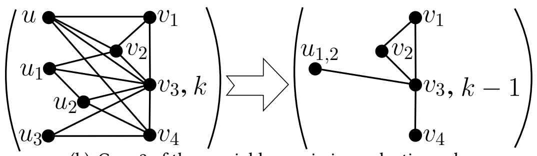
A APPENDIX

A.1 Three Neighbors Missing Reduction Rules

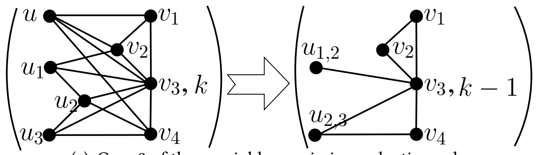
The four cases of the three neighbors missing reduction rules are illustrated in Figure 7.



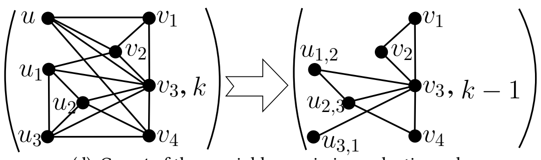
(a) Case 1 of three neighbors missing reduction rule



(b) Case 2 of three neighbors missing reduction rule



(c) Case 3 of three neighbors missing reduction rule



(d) Case 4 of three neighbors missing reduction rule

Figure 7: Three neighbors missing reduction rules

A.2 Pseudocode of MC-DD

The pseudocode of MC-DD is shown in Algorithm 5. We first compute a clique C^* by using the maximum degree-based heuristic (Line 1). Specifically, given an initial clique C consisting of one vertex, we iteratively add to C the vertex that has the largest degree and is adjacent to all vertices of C. To obtain a larger clique, we run this process for the 10 vertices with the largest degrees, and then choose the largest clique. Then, we reduce the input graph G to its $|C^*|$-core for time efficiency (Line 2), compute a degeneracy ordering Dorder$(\cdot)$ for G (Line 3), and obtain the degeneracy-based clique of G (Line 4). In order to compute an upper bound of the clique number of G, we also compute a graph coloring color$(\cdot)$ and core numbers core$(\cdot)$ of $V(G)$ (Line 6); note that, here we do not use the recoloring technique. The upper bound then is obtained as the number of distinct colors used in the coloring (Line 7).

Algorithm 5: MC-DD($G = (V, E)$)

```
1 Compute a clique $C^*$ by the maximum degree-based heuristic;
2 $G \leftarrow$ the $|C^*|$-core of $G$;
3 Compute a degeneracy ordering Dorder$(\cdot)$ of $V(G)$;
4 $C \leftarrow$ the degeneracy-based clique of $G$;
5 if $|C| > |C^*|$ then $C^* \leftarrow C$;
6 Compute color$(\cdot)$ and core$(\cdot)$ for vertices of $V(G)$;
7 $\overline{\omega} \leftarrow$ UniqColors($V(G)$, color$(\cdot)$);
8 return ($C^*$, $\overline{\omega}$, Dorder$(\cdot)$, color$(\cdot)$, core$(\cdot)$);
```

A.3 Additional Experimental Information

Statistics of the tested graphs are shown in Table 5, where graphs are in increasing order regarding the number of edges. Here, the number of edges is the number of *undirected* edges.

Table 5: Statistics of real graphs (density: $\frac{2m}{n(n-1)}$)

Graph	#Vertices	#Edges	density	Type
Email	224,832	339,925	$< 10^{-4}$	Commun.
Epinions	75,877	405,739	$< 10^{-3}$	Social
slashdot	77,350	468,554	$< 10^{-3}$	Social
DBLP	317,080	1,049,866	$< 10^{-4}$	Collabor.
Amazon	403,364	2,443,311	$< 10^{-4}$	Copurchase
Google	875,713	4,322,051	$< 10^{-4}$	Web
wiki-Talk	2,388,953	4,656,682	$< 10^{-5}$	Commun.
human-gene2	14,022	9,027,024	0.092	Biological
as-Skitter	1,694,616	11,094,209	$< 10^{-5}$	Auto. Sys
human-gene1	21,890	12,323,680	0.052	Biological
soc-flickr	1,715,255	15,555,041	$< 10^{-4}$	Social
patent	3,774,768	16,518,947	$< 10^{-5}$	Citation
soc-pokec	1,632,803	22,301,964	$< 10^{-4}$	Social
LiveJ	4,843,953	42,845,684	$< 10^{-5}$	Social
twitter-mpi	9,862,152	99,940,317	$< 10^{-5}$	Social
tech-p2p	5,792,297	147,829,887	$< 10^{-5}$	Tech.
uk-2002	18,459,128	261,556,721	$< 10^{-5}$	Web
webbase	115,554,441	854,809,761	$< 10^{-6}$	Web
it-2004	41,290,577	1,027,474,895	$< 10^{-5}$	Web

To store large real graphs with hundreds of millions of vertices in main memory, we use the *adjacency array* representation (aka *compressed sparse row* representation [5]). That is, we store the set of neighbors of a vertex consecutively in a single large adjacency array, whereas its start position is stored in another array of size n. As a result, the input graph G is represented by $2m + n$ integers.