Efficient Maximum k-Defective Clique Computation with Improved Time Complexity

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

ABSTRACT

k-defective cliques relax cliques by allowing up-to k missing edges from being a complete graph. This relaxation enables us to find larger near-cliques and has applications in link prediction, cluster detection, social network analysis and transportation science. The problem of finding the largest k-defective clique has been recently studied with several algorithms being proposed in the literature. However, the currently fastest algorithm KDBB does not improve its time complexity from being the trivial $O(2^n)$, and also, KDBB's practical performance is still not satisfactory. In this paper, we advance the state of the art for exact maximum k-defective clique computation, in terms of both time complexity and practical performance. Moreover, we separate the techniques required for achieving the time complexity from others purely used for practical performance consideration; this design choice may facilitate the research community to further improve the practical efficiency while not sacrificing the worst case time complexity. In specific, we first develop a general framework kDC that beats the trivial time complexity of $O(2^n)$ and achieves a better time complexity than all existing algorithms. The time complexity of kDC is solely achieved by our newly designed non-fully-adjacent-first branching rule, excess-removal reduction rule and high-degree reduction rule. Then, to make kDC practically efficient, we further propose a new upper bound, two new reduction rules, and an algorithm for efficiently computing a large initial solution. Extensive empirical studies on three benchmark graph collections with 290 graphs in total demonstrate that kDC outperforms the currently fastest algorithm KDBB by several orders of magnitude.

CCS CONCEPTS

• **Mathematics of computing → Graph algorithms**; • **Information systems → Social networks**.

ACM Reference Format:
Lijun Chang. 2023. Efficient Maximum k-Defective Clique Computation with Improved Time Complexity. *Proc. ACM Manag. Data* 1, 3 (SIGMOD), Article 209 (September 2023), 14 pages. https://doi.org/10.1145/3617313

Author's address: Lijun Chang, Lijun.Chang@sydney.edu.au, The University of Sydney, Sydney, Australia.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
2836-6573/2023/9-ART209 $15.00
https://doi.org/10.1145/3617313

1 INTRODUCTION

The relationship among entities in many applications, such as social media, communication networks, collaboration networks, web graphs, and the Internet, can be naturally captured by the graph model. As a result, real-world graph data is abundant, and graph-based data analysis has been widely used to extract insights for guiding the decision-making process. In particular, the problem of identifying dense (i.e., cohesive) subgraphs has been extensively studied [10, 24], since it serves many applications. For example, identifying large dense subgraphs has been used for detecting anomalies in financial networks [2], identifying real-time stories in social media [3], detecting communities in social networks [5], and finding protein complexes in biological networks [40].

Clique (i.e., complete subgraph) is a classic notion for defining dense subgraphs, which requires every pair of distinct vertices in the subgraph to be directly connected by an edge. It is easy to see that a clique is the densest structure that a subgraph can be. As a result, clique related problems have been extensively explored in the literature, and many advancements have been made regarding clique computation. For example, it has been shown that the maximum clique is not only NP-hard to compute exactly [23], but also NP-hard to approximate within a factor of $n^{1-\epsilon}$ for any constant $0 < \epsilon < 1$ [19]; here n denotes the number of vertices in the input graph G. Nevertheless, exact algorithms have been studied both theoretically and practically in the literature. The state-of-the-art time complexity for maximum clique computation is $O^*(1.1888^n)$ [33], and one of the practically efficient algorithms is MC-BRB [8]; here the O^* notation hides polynomial factors. In addition, the problems of enumerating all maximal cliques, enumerating all cliques of the maximum size, and enumerating and counting all cliques with k vertices for a small k have also been extensively studied [9, 14, 20, 27, 42].

Requiring a large subgraph to be fully connected however is often too restrictive for many applications, such as complex network analysis [31], considering that data is often noisy or incomplete. Hence, various clique relaxations have been formulated in the literature, such as quasi-clique [1], k-plex [4], k-club [6], and k-defective clique [49]. In this paper, we focus on the k-defective clique, which allows a subgraph to miss up-to k edges to be a complete subgraph; note that a k-defective clique for $k = 0$ is a clique. The concept of k-defective clique was formulated in [49] for predicting missing interactions between proteins in biological networks. Besides, it also finds applications in cluster detection [39], transportation science [38], and social network analysis [17, 21]. Since a clique is also a k-defective clique for any $k \geq 0$, the maximum k-defective clique is no less than and usually can be much larger than the maximum clique. Consider the graph in Figure 1, it is easy to see that the maximum clique size is 4, while the maximum k-defective clique size for any $k \leq 4$ is $4 + k$; specifically, the entire graph is a 4-defective clique, and the remaining graph after removing any vertex is a 3-defective clique.

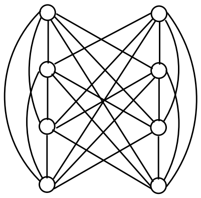


Figure 1: Clique vs. k-Defective Clique

The problem of maximum k-defective clique computation is also NP-hard [48]. The state-of-the-art time complexity for maximum k-defective clique computation that beats the trivial $O^*(2^n)$ time complexity is achieved by the MADEC$^+$ algorithm proposed in [11], which runs in $O^*(\sigma_k^n)$ time where $\sigma_k < 2$ is the largest real root of the equation $x^{2k+3} - 2x^{2k+2} + 1 = 0$. Although a graph coloring-based upper bound as well as other pruning techniques are proposed in [11] aiming to improve the practical performance of MADEC$^+$, it is shown in [16] that the graph coloring-based upper bound proposed in [16] is ineffective and MADEC$^+$ is inefficient in practice especially when $k \geq 10$. For example, for $k \geq 15$ on the Facebook graphs collection (please refer to Section 4 for the description of the dataset), even the version of MADEC$^+$ that is further optimized by the authors of [16] was still not able to find the maximum k-defective clique for any graph instance with a time limit of 3 hours. With the goal of enhancing the practical performance, the KDBB algorithm is designed in [16] which proposes and incorporates preprocessing as well as multiple pruning techniques. Nevertheless, KDBB is still inefficient, and moreover, no time complexity better than $O^*(2^n)$ has been proved for KDBB.

In this paper, we aim to advance the state of the art for maximum k-defective clique computation, both theoretically and practically. We first develop a general backtracking framework kDC based on our newly designed non-fully-adjacent-first branching rule (**BR**), excess-removal reduction rule (**RR1**) and high-degree reduction rule (**RR2**). We prove that our framework runs in $O^*(\gamma_k^n)$ time where $\gamma_k < 2$ is the largest real root of the equation $x^{k+3} - 2x^{k+2} + 1 = 0$. In comparison, the time complexity of MADEC$^+$ is $O^*(\gamma_{2k}^n)$ by observing that $\sigma_k = \gamma_{2k}$. Note that $\gamma_k < \gamma_{2k}$. Thus, we advance the state of the art regarding the theoretical time complexity. We remark that the time complexity of kDC is solely achieved by our branching rule **BR** and reduction rules **RR1** and **RR2**, and these are the minimal requirements for achieving the time complexity of $O^*(\gamma_k^n)$. We deliberately separate the techniques required for achieving the time complexity from the ones used purely for improving the practical performance, such that others may further improve the efficiency while retaining the time complexity of $O^*(\gamma_k^n)$.

To make kDC practically efficient, we further propose techniques from three aspects: an improved graph coloring-based upper bound (**UB1**), a degree-sequence-based reduction rule (**RR3**), a second-order reduction rule (**RR4**), and a new algorithm Degen-opt for efficiently computing a large initial solution. Specifically, given a graph g and a k-defective clique represented by its set of vertices S such that $S \subseteq V(g)$, our improved coloring-based upper bound **UB1** computes an upper bound of the largest k-defective clique that is in g and contains S, and prunes the backtracking instance (g, S) if the computed upper bound is no larger than the currently found largest solution. Same as the graph coloring-based upper bound proposed in [11], **UB1** also utilizes graph coloring; but **UB1** computes a much tighter (i.e., smaller) upper bound than [11]. In essence, a (greedy) graph coloring is used to partition the vertices into independent sets, by observing that all vertices with the same color form an independent set. Let $\pi_1, \ldots, \pi_c$ be a partitioning of $V(g) \setminus S$ into independent sets. The upper bound computed in [11] is $|S| + \sum_{i=1}^{c} \min(\lfloor \frac{1+\sqrt{8k+1}}{2} \rfloor, |\pi_i|)$, which is based on the observation that an independent set with more than $\lfloor \frac{1+\sqrt{8k+1}}{2} \rfloor$ vertices will miss more than k edges and thus cannot be all contained in the same k-defective clique. The upper bound of [11] has two deficiencies. Firstly, it allows c independent sets, each of size up-to $\lfloor \frac{1+\sqrt{8k+1}}{2} \rfloor$, to be included into the solution for computing the upper bound; this will actually introduce almost $c \times k$ missing edges, much larger than the allowed k missing edges. In particular, if $|\pi_i| \geq \lfloor \frac{1+\sqrt{8k+1}}{2} \rfloor$ for each $1 \leq i \leq c$, then the upper bound becomes $|S| + c \times \lfloor \frac{1+\sqrt{8k+1}}{2} \rfloor$, while $|S| + c + k$ is a much smaller upper bound. Secondly, it ignores the missing edges within S and the missing edges between S and $V(g) \setminus S$; for example, the upper bound remains the same even if S already has k missing edges. Our **UB1** computes a tighter upper bound than $|S| + c + k - |\overline{E}(S)|$ by resolving the above two issues; here $\overline{E}(S)$ denotes the set of missing edges in S.

Contributions. Our main contributions are as follows.

- We develop a general framework kDC for computing the maximum k-defective clique in $O^*(\gamma_k^n)$ time, based on our newly designed branching rule **BR** and reduction rules **RR1** and **RR2**; here $\gamma_k < 2$ is the largest real root of the equation $x^{k+3} - 2x^{k+2} + 1 = 0$. (Section 3.1)
- We propose a new upper bound **UB1** based on graph coloring, which can be computed in linear time and is much tighter than the upper bounds proposed in [11, 16]. (Section 3.2.1)
- We propose two new reduction rules **RR3** and **RR4** that can be conducted in linear time. (Section 3.2.2)
- We propose an algorithm Degen-opt for computing a large initial k-defective clique in $O(\delta(G) \times m)$ time, where m is the number of edges and $\delta(G) \leq \sqrt{m}$ is the degeneracy of G. (Section 3.3)

We also conduct extensive empirical studies on three benchmark graph collections with 290 graph instances in total to evaluate our techniques (Section 4). The results show that (1) on the real-world graphs collection, kDC with a time limit of 3 *seconds* solves even more graph instances than the existing fastest algorithm KDBB with a time limit of 3 *hours*, and (2) on the 41 Facebook graphs that have more than 15,000 vertices, kDC is on average three orders of magnitude faster than KDBB. In addition, our ablation studies demonstrate that each of our additional techniques (i.e., upper bound **UB1**, reduction rules **RR3** and **RR4**, and initial solution computation) improves the practical efficiency of kDC.

2 PRELIMINARIES

In this paper, we focus on a large *unweighted* and *undirected* graph $G = (V, E)$, where V is the set of vertices and E is the set of undirected edges; we consider only *simple* graphs, i.e., without self-loops and parallel edges. Let $n = |V|$ and $m = |E|$ denote the cardinalities of V and E, respectively. We denote the undirected edge between u and v by both (u, v) and (v, u); then, 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, i.e., $G[S] = (S, \{(u, v) \in E \mid u, v \in S\})$. For ease of presentation, we simply refer to an unweighted and undirected graph as a graph, and omit the subscript G from the notations when the context is clear. For an arbitrary given graph g, we denote its set of vertices and its set of edges by $V(g)$ and $E(g)$, respectively.

Definition 2.1 (Clique). A graph g is a clique (i.e., complete graph) if it has an edge between every pair of distinct vertices, i.e., $|E(g)| = \frac{|V(g)|(|V(g)|-1)}{2}$ or equivalently, $d_g(u) = |V(g)| - 1, \forall u \in V(g)$.

Definition 2.2 (k-Defective Clique). A graph g is a k-defective clique if it misses at most k edges, i.e., $|E(g)| \geq \frac{|V(g)|(|V(g)|-1)}{2} - k$.

The definition of k-defective clique relaxes the definition of clique by allowing a few (i.e., k) missing edges, and 0-defective cliques are cliques. Obviously, if a subgraph g of G is a k-defective clique, then the subgraph of G induced by vertices $V(g)$ is also a k-defective clique. Thus, in this paper, ***we simply refer to a k-defective clique by its set of vertices***, and measure the size of a k-defective clique $C \subseteq V$ by the number of vertices, i.e., $|C|$.

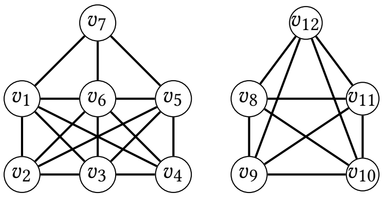


Figure 2: An example graph

The property of k-defective clique is *hereditary*, i.e., any subset of a k-defective clique is also a k-defective clique. A k-defective clique C of G is a *maximal k-defective clique* if every proper superset of C in G is not a k-defective clique, and is a *maximum k-defective clique* if its size is the largest among all k-defective cliques of G; note that the maximum k-defective clique is not unique. Consider the graph in Figure 2, $\{v_8, v_9, \ldots, v_{12}\}$ is a maximum clique and is also a maximum 1-defective clique. In addition, both $\{v_1, v_2, v_3, v_4, v_6\}$ and $\{v_1, v_2, v_3, v_5, v_6\}$ are maximum 1-defective cliques that miss the edge (v_2, v_4) and the edge (v_1, v_5), respectively. $\{v_1, v_2, \ldots, v_6\}$ is a maximum 2-defective clique that misses edges (v_2, v_4) and (v_1, v_5).

To facilitate the presentation, we denote the set of edges that are missing from a graph g by $\overline{E}(g)$, i.e., $(u, v) \in \overline{E}(g)$ if and only if $u \neq v$ and $(u, v) \notin E(g)$; we call the edges of $\overline{E}(g)$ as ***non-edges*** of g. Thus, g is a k-defective clique if and only if $|\overline{E}(g)| \leq k$. For two vertices u and v that are not adjacent (i.e., not connected by an edge), we call v (resp. u) a ***non-neighbor*** of u (resp. v); note that ***a vertex is considered neither a neighbor nor a non-neighbor of itself***. We denote the set of all non-neighbors of u in G by $\overline{N}_G(u) = V(G) \setminus (N_G(u) \cup u)$; note that, for presentation simplicity, we denote the union of a set S and a vertex u by $S \cup u$, and denote the subtraction of u from S by $S \setminus u$. For any set S of vertices and a vertex u (where u could be either in or not in S), we abbreviate $N_{G[S \cup u]}(u)$ as $N_S(u)$ and abbreviate $\overline{N}_{G[S \cup u]}(u)$ as $\overline{N}_S(u)$.

Table 1: Frequently used notations

Notation	Meaning
$G = (V, E)$	an unweighted and undirected graph with vertex set V and edge set E
$g = (V(g), E(g))$	a subgraph of G
$S, C \subseteq V$	k-defective cliques
$N_S(u)$	the set of u's *neighbors* that are in S
$\overline{N}_S(u)$	the set of u's *non-neighbors* that are in S
$d_S(u)$	the *number* of u's neighbors that are in S
$E(S)$	the set of (undirected) *edges* in the subgraph of g (or G) induced by S
$\overline{E}(S)$	the set of (undirected) *non-edges* in the subgraph of g (or G) induced by S

Problem Statement. Given a graph $G = (V, E)$ and an integer $k \geq 1$, we study the problem of maximum k-defective clique computation, aiming to find the largest k-defective clique in G.

Frequently used notations are summarized in Table 1.

2.1 Degeneracy Ordering, k-Core and k-Truss

In this subsection, we review the concepts of degeneracy ordering, k-core and k-truss, which will be used in Section 3.2.3.

Definition 2.3 (Degeneracy ordering). Given a graph G, an ordering $(v_1, v_2, \ldots, v_n)$ of its vertices is a degeneracy ordering if for each $1 \leq i \leq n$, v_i is the vertex with the smallest degree in the subgraph of G induced by vertices $\{v_i, v_{i+1}, \ldots, v_n\}$.

Definition 2.4 (k-core [37]). Given a graph G and an integer k, the k-core of G is the maximal subgraph g of G such that every vertex $u \in V(g)$ has degree $d_g(u) \geq k$ in the subgraph g.

k-core is a *vertex-induced* subgraph. The degeneracy ordering can be computed in $O(m)$ time by the peeling algorithm [10, 28], which iteratively removes the vertex with *the smallest degree* from the graph and appends it to the end of the ordering. Note that, although the k-core can also be computed by the peeling algorithm, it is usually more efficient to directly compute the k-core by iteratively removing vertices of *degree smaller than k* from the graph [10]. The largest k such that G contains a non-empty k-core is known as the ***degeneracy*** of G, denoted $\delta(G)$; note that $\delta(G) \leq \sqrt{m}$ [14]. For the graph in Figure 2, $(v_7, v_1, v_2, v_3, v_4, v_5, v_6, v_8, v_9, v_{10}, v_{11}, v_{12})$ is a degeneracy ordering. The entire graph is a 3-core, and the subgraph obtained by removing v_7 is a 4-core; $\delta(G) = 4$, as it has no 5-core.

Definition 2.5 (k-truss [46]). Given a graph G and an integer k, the k-truss of G is the maximal subgraph g of G such that every edge $(u, v) \in E(g)$ participates in at least $k - 2$ triangles, i.e., $|N_g(u) \cap N_g(v)| \geq k - 2, \forall (u, v) \in E(g)$.

k-truss is a subgraph of the $(k-1)$-core, and is an *edge-induced* subgraph. k-truss can be considered as a higher-order version of k-core. That is, each edge corresponds to a node, and each triangle corresponds to a hyper-edge, in a hyper-graph. Hence, the k-truss can be computed in a similar way to k-core, but the time complexity becomes $O(\delta(G) \times m)$ [46]. For the graph in Figure 2, the entire graph is a 3-truss, the subgraph obtained by removing edges $\{(v_7, v_1), (v_7, v_6), (v_7, v_5)\}$ (and thus also vertex v_7) is a 4-truss, and the subgraph induced by vertices $\{v_8, v_9, \ldots, v_{12}\}$ is a 5-truss which is contained in the 4-core.

3 OUR APPROACH

In this section, we propose an efficient algorithm kDC for exact maximum k-defective clique computation. As the maximum k-defective clique computation problem is NP-hard [48], our algorithm kDC, as well as all other exact algorithms, will run in exponential time in the worst case. Nevertheless, our algorithm kDC beats the trivial time complexity of $O^*(2^n)$ where the O^* notation hides polynomial factors. Specifically, we prove that our algorithm kDC runs in $O^*(\gamma_k^n)$ time where $\gamma_k < 2$ is the largest real root of the equation $x^{k+3} - 2x^{k+2} + 1 = 0$; note that this improves the state-of-the-art time complexity $O^*(\gamma_{2k}^n)$ [11], as γ_k increases regarding k (i.e., $\gamma_k < \gamma_{2k}$).

In the following, we first in Section 3.1 present the framework of kDC and prove its time complexity. Then, we in Section 3.2 propose upper bounds and reduction rules to improve the practical performance of kDC. Lastly, we in Section 3.3 present a heuristic algorithm for initially computing a large k-defective clique.

3.1 The Framework of kDC

Our algorithm falls into the category of branch-and-bound search (also known as backtracking) algorithms; we will use the terms *backtracking* and *branch-and-bound search* interchangeably. The general idea is as follows. Let (g, S) denote an instance of the backtracking, where g is a graph and $S \subseteq V(g)$ is a k-defective clique in g. The goal of solving an instance is to find the largest k-defective clique in the instance; *a k-defective clique is said to be in the instance (g, S) if it is in g and contains S*. To solve the instance (g, S), a backtracking algorithm will select a branching vertex $b \in V(g) \setminus S$, and then recursively solve two new instances that are generated based on b: one instance includes b into S, and the other removes b from g (and thus excludes b from being added into S). Solving the instance $(G, \emptyset)$ thus finds the maximum k-defective clique in G. All the instances that are generated in solving the instance $(G, \emptyset)$ form a binary search tree, a snippet of which is shown in Figure 3. Each node of the search tree, denoted by I_i, represents an instance of the backtracking (i.e., $I_i = (g_i, S_i)$), and has two children representing the two new instances that are generated based on the branching vertex of the instance I_i. For example, in Figure 3, the branching vertex selected for the instance I_0 is b_0, and the two new instances that are generated based on b_0 are I_1 (which includes b_0 into the solution) and I_{q+1} (which removes b_0 from the graph); the actions of including b_0 and removing b_0 are, respectively, represented as labels $+b_0$ and $-b_0$ on the corresponding edges in the search tree.

Backtracking algorithms differ from each other in three aspects:

- **Branching techniques** determine which vertex is selected as the branching vertex, e.g., b_0 for the instance I_0 and b_1 for the instance I_1 in Figure 3.
- **Reducing techniques** reduce the size of an instance, i.e., transform an instance (g, S) to another equivalent instance (g', S') with $|V(g') \setminus S'| \leq |V(g) \setminus S|$.[1]
- **Upper bounding techniques** prune an instance, as well as the entire search subtree rooted at the instance, if a computed upper bound of the largest k-defective clique in the instance is no larger than the best solution found so far.

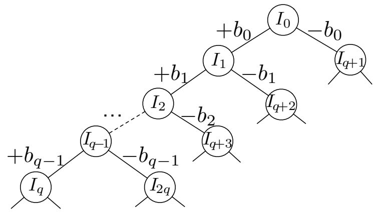


Figure 3: A snippet of the (binary) search tree $\mathcal{T}$ of a backtracking algorithm

In this paper, we propose new techniques from all the three aspects for the problem of maximum k-defective clique computation. In this subsection, we only present the techniques that are required to achieve our time complexity of $O^*(\gamma_k^n)$, and defer other practical techniques to Sections 3.2 and 3.3.

3.1.1 Techniques for Achieving Our Time Complexity. We first propose the following non-fully-adjacent-first branching rule (**BR**), which prefers branching on a vertex that is not fully adjacent to S.

BR (non-fully-adjacent first branching rule). Given an instance (g, S), the branching vertex is selected as the one of $V(g) \setminus S$ that has at least one non-neighbor in S; if all vertices of $V(g) \setminus S$ are adjacent to all vertices of S, then the branching vertex is an arbitrary vertex of $V(g) \setminus S$.

Note that, the way of selecting a branching vertex will not compromise the correctness of the algorithm, *as long as the union of S and the branching vertex forms a k-defective clique.* To achieve our time complexity, we also propose the following two reduction rules.

RR1 (excess-removal reduction rule). Given an instance (g, S), for a vertex $u \in V(g) \setminus S$ satisfying $|\overline{E}(S \cup u)| > k$, we remove u from g.

RR2 (high-degree reduction rule). Given an instance (g, S), for a vertex $u \in V(g) \setminus S$ satisfying $|\overline{E}(S \cup u)| \leq k$ and $d_g(u) \geq |V(g)| - 2$, we greedily add u to S.

The reduction rule **RR1** ensures that the union of S and any branching vertex (including the one selected by our branching rule **BR**) form a valid k-defective clique, by noting that *all reduction rules are applied before the branching rule.* The correctness of the reduction rule **RR1** is trivial, and we prove the correctness for the reduction rule **RR2** in the lemma below.

Lemma 3.1. *Given an instance (g, S), for a vertex $u \in V(g) \setminus S$ satisfying $|\overline{E}(S \cup u)| \leq k$ and $d_g(u) \geq |V(g)| - 2$, there is a maximum k-defective clique in the instance that contains u.*

Proof. The case of $d_g(u) = |V(g)| - 1$ (i.e., u is adjacent to all other vertices in g) is trivial. Let's focus on the case of $d_g(u) = |V(g)| - 2$ and consider a maximum k-defective clique C in the instance that does not contain u, i.e., $u \notin C$ and $S \subseteq C \subseteq V(g)$. Let v be the unique non-neighbor of u in g. Then, v must be in C, as otherwise $C \cup u$ would be a k-defective clique of size larger than C. We consider two cases depending on whether $v \in S$.

Case-I: $v \notin S$. That is, $v \in C \setminus S$. It is easy to verify that $C \cup u \setminus v$ is a k-defective clique of the same size as C and contains u and S.

[1] Note that, reducing techniques could also remove edges from the graph, e.g., the reduction rule **RR6** in Section 3.2.2. We omit the discussions here for simplicity.

Case-II: $v \in S$. There must exist a vertex of $C \setminus S$ that has at least one non-neighbor in C, since otherwise $C \cup \{u\}$ would also be a valid k-defective clique by noting that $|\overline{E}(S \cup u)| \leq k$; let w be such a vertex of $C \setminus S$. It is easy to verify that $C \cup u \setminus w$, which contains u and S, is a k-defective clique of the same size as C. □

Algorithm 1: kDC-t(G, k)

```
  Input: A graph $G$ and an integer $k$
  Output: A maximum $k$-defective clique $C^*$ of $G$
1 $C^* \leftarrow \emptyset$;
2 Branch&Bound-t$(G, \emptyset)$;
3 return $C^*$;
  Procedure Branch&Bound-t$(g, S)$
4 $(g', S') \leftarrow$ apply reduction rules RR1 and RR2 to $(g, S)$;
5 if $g'$ is a $k$-defective clique then update $C^*$ by $V(g')$ and return;
6 $b \leftarrow$ a vertex of $V(g') \setminus S'$ that has at least one non-neighbor in $S'$;
    /* If there is no such a vertex, then $b$ is an arbitrary
    vertex of $V(g') \setminus S'$ */;
7 Branch&Bound-t$(g', S' \cup b)$;   /* Left branch includes $b$ */;
8 Branch&Bound-t$(g' \setminus b, S')$;   /* Right branch excludes $b$ */;
```

Based on the above discussions, the pseudocode of our algorithm kDC-t is shown in Algorithm 1; here t stands for theoretical as the algorithm only considers the theoretical aspect. It takes a graph G and an integer k as input, and outputs a maximum k-defective clique C^* of G which is achieved by recursively invoking Branch&Bound-t to grow a partial solution S that is initialized as $\emptyset$ (Line 2). In the procedure Branch&Bound-t, we first apply reduction rules **RR1** and **RR2** to reduce the instance (g, S) to a potentially smaller instance (g', S') satisfying $V(g') \subseteq V(g)$ and $S' \supseteq S$ (Line 4). If g' itself is a k-defective clique, then we update the currently found largest k-defective clique C^* by $V(g')$ and backtrack (Line 5). Otherwise, we pick a branching vertex b based on our branching rule **BR** (Line 6), and then generate two new instances of Branch&Bound-t and go into recursion (Lines 7–8).

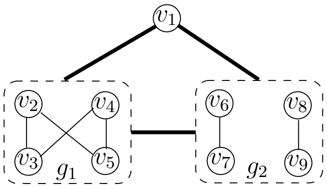


Figure 4: Running example for Algorithm 1 (thick edge indicates full connection between the subgraphs)

Example 3.2. Consider the example graph in Figure 4 where thick edge indicates full connection between the corresponding subgraphs, i.e., v_1 is adjacent to every other vertex, and every vertex of g_1 is adjacent to every vertex of g_2. Suppose $k = 3$, when invoking Branch&Bound-t with $g = G$ and $S = \emptyset$, our reduction rule **RR2** will greedily and iteratively move v_1, v_2, v_3, v_4, v_5 to S. Then, an arbitrary vertex of $\{v_6, \ldots, v_9\}$ can be selected as the branching vertex; suppose v_6 is selected. The newly generated left branch would have $S_1 = \{v_1, \ldots, v_6\}$, and the reduction rules **RR1** and **RR2** would have no effect for S_1; note that the graph g will remain unchanged in the remaining part of this example. The branching vertex selected for S_1 could be either v_8 or v_9 (as they are not fully adjacent to S_1) but not v_7 (which is fully adjacent to S_1); suppose v_8 is selected. The newly generated left branch for S_1 would have $S_2 = \{v_1, \ldots, v_6, v_8\}$ which contains three non-edges, and thus the reduction rule **RR1** will remove v_7 and v_9 from the graph.

3.1.2 *Time Complexity Analysis of Algorithm 1.*

To analyze the time complexity of Algorithm 1, we consider the search tree $\mathcal{T}$ of recursively invoking Branch&Bound-t, as shown in Figure 3. To avoid confusion, we refer to nodes of the search tree by *nodes*, and vertices of a graph by *vertices*. Recall that each node of $\mathcal{T}$ represents an instance of Branch&Bound-t, i.e., (g, S), and has two children: the left child includes the branching vertex b to S, and the right child excludes b from g. It is worth mentioning that each child may also include or exclude other vertices due to applying reduction rules **RR1** and **RR2**. We use $I, I', I_0, I_1, \ldots$ to denote nodes of $\mathcal{T}$, and use $I.g$ and $I.S$ to respectively denote the graph g and the partial solution S of the Branch&Bound-t instance to which I corresponds. We would like to emphasize that $I.g$ ***and*** $I.S$ ***denote the ones obtained after applying the reduction rules*** at Lines 4–5 of Algorithm 1, not the ones input to Branch&Bound-t; note that Line 5 can be regarded as applying the following reduction rule:

- If g' is a k-defective clique, then all vertices of $V(g') \setminus S'$ are moved to S'.

In this case, the instance will not generate any children (i.e., any new instances) and thus becomes a leaf node. We measure the size of I by the number of vertices in the graph $I.g$ that are not in the partial solution $I.S$, i.e., $|I| = |V(I.g)| - |I.S| = |V(I.g) \setminus I.S| \geq 0$. It is easy to see that $|I'| \leq |I| - 1$ *whenever* I' *is a child of* I *— e.g., the branching vertex* b *of* I *is in* $V(I.g) \setminus I.S$ *but not in* $V(I'.g) \setminus I'.S$ *— and* $|I| = 0$ *whenever* I *is a leaf node.*

Before proving the time complexity, we first state the following important property of exhaustively applying the reduction rules **RR1** and **RR2**, whose proof is omitted due to space limitation.

Lemma 3.3. *After exhaustively applying the reduction rules* ***RR1*** *and* ***RR2****, the resulting instance* (g, S) *satisfies the following condition:*

- *For every vertex* $u \in V(g) \setminus S$*, it holds that* $|\overline{E}(S \cup u)| \leq k$ *and* $d_g(u) < |V(g)| - 2$.

i.e., ***all vertices of*** $V(g) \setminus S$ ***have at least two non-neighbors in*** g.

We are now ready to prove the time complexity of Algorithm 1 in the following lemma and theorem.

Lemma 3.4. *Let* $\mathcal{T}$ *be the search tree of running Algorithm 1 (i.e., recursively invoking* Branch&Bound-t*). For any node* I *of* $\mathcal{T}$*, the number of leaf nodes in the subtree of* $\mathcal{T}$ *rooted at* I*, denoted* $\ell(I)$*, is at most* $\gamma_k^{|I|}$*, where* $1 < \gamma_k < 2$ *is the largest real root of the equation* $x^{k+3} - 2x^{k+2} + 1 = 0$.

Proof. We prove the lemma by induction. For the base case that I is a leaf node, it is trivial that $\ell(I) = 1 \leq \gamma_k^{|I|}$ since $\gamma_k > 1$ and $|I| = 0$. For a non-leaf node I, for any path $(I_0 = I, I_1, \ldots, I_{q-1}, I_q)$ with $q \geq 1$ that starts from I and always visits the left child in the search tree $\mathcal{T}$, it is trivial that

$$\ell(I) = \ell(I_{q+1}) + \ell(I_{q+2}) + \cdots + \ell(I_{2q}) + \ell(I_q)$$

here, $I_{q+1}, I_{q+2}, \ldots, I_{2q}$ are the right child of $I_0, I_1, \ldots, I_{q-1}$, respectively, as illustrated in Figure 3. To bound $\ell(I)$, let's specifically consider the path $(I_0 = I, I_1, \ldots, I_q)$ where I_q is the first node such that $|I_q| \leq |I_{q-1}| - 2$; this implies that for $1 \leq i < q$, $|I_i| = |I_{i-1}| - 1$ and consequently $V(I_i.g) = V(I.g)$ and the reduction rules at Line 4 of Algorithm 1 have no effect on I_{i-1}. Note that such a node I_q always exists since (1) $I.g$ is not a k-defective clique (otherwise, I would be a leaf node) and (2) a leaf node I' satisfies $|I'| = 0$ (i.e., $I'.S = V(I'.g) \neq V(I.g)$ and thus I' would satisfy the condition). Hence, the following two facts hold.

Fact 1. $|I_i| \leq |I_{i-q-1}| - 1 \leq |I| + q - i$, for $q + 1 \leq i \leq 2q$.

Fact 2. $|I_q| \leq |I_{q-1}| - 2 \leq |I| - q - 1$.

Now, we prove that the following fact also holds.

Fact 3. $q \leq k + 1$.

We prove Fact 3 by contradiction. Suppose $q \geq k+2$. Let I_x be the last node, on the path $(I_0 = I, I_1, \ldots, I_x, \ldots, I_q)$, satisfying the condition that all vertices of $V(I_x.g) \setminus I_x.S$ are adjacent to all vertices of $I_x.S$, i.e., the branching vertex b_x selected for I_x has no non-neighbor in $I_x.S$; without loss of generality, we assume such an I_x exists, otherwise the following proof still holds by setting $x = 0$. Then, $|\overline{E}(I_x.S)| \geq x$ since (1) each branching vertex added to $I_x.S$ along the path $(I_0 = I, \ldots, I_{x-1})$ has at least two non-neighbors (according to Lemma 3.3), and (2) all these non-neighbors are in $I_x.S$ (according to the definition of I_x); note that, according to the definition of I_q, the reduction rules have no effect on I_i for $1 \leq i < q$. This implies that $x \leq k$. Then, according to the definition of I_x and our branching rule **BR**, for each i with $x + 1 \leq i < q$, the branching vertex selected for I_i has at least one non-neighbor in $I_i.S$, and consequently,

$$|\overline{E}(I_q.S)| \geq |\overline{E}(I_x.S)| + (q - x - 1) \geq q - 1 \geq k + 1$$

contradicting that $I_q.S$ is a k-defective clique. Hence, Fact 3 holds.

Based on Facts 1, 2 and 3, we have

$$\begin{aligned} \ell(I) &= \ell(I_{q+1}) + \ell(I_{q+2}) + \cdots + \ell(I_{2q}) + \ell(I_q) \\ &\leq \gamma_k^{|I_{q+1}|} + \gamma_k^{|I_{q+2}|} + \cdots + \gamma_k^{|I_{2q}|} + \gamma_k^{|I_q|} \\ &\leq \gamma_k^{|I|-1} + \gamma_k^{|I|-2} + \cdots + \gamma_k^{|I|-q} + \gamma_k^{|I|-q-1} \\ &\leq \gamma_k^{|I|-1} + \gamma_k^{|I|-2} + \cdots + \gamma_k^{|I|-k-1} + \gamma_k^{|I|-k-2} \end{aligned} \tag{1}$$

where $\gamma_k^{|I|-1} + \gamma_k^{|I|-2} + \cdots + \gamma_k^{|I|-k-1} + \gamma_k^{|I|-k-2} \leq \gamma_k^{|I|}$ if γ_k is no smaller than the largest real root of the equation $x^{k+2} - x^{k+1} - \cdots - x - 1 = 0$ which is equivalent to the equation $x^{k+3} - 2x^{k+2} + 1 = 0$ [15]. The first few solutions to the equation are $\gamma_0 = 1.619$, $\gamma_1 = 1.840$, $\gamma_2 = 1.928$, $\gamma_3 = 1.966$, $\gamma_4 = 1.984$ and $\gamma_5 = 1.992$. □

Theorem 3.5. *Let P_1 be the time complexity of running Lines 4–6 of Algorithm 1, which is polynomial in n. Then, the time complexity of Algorithm 1 is $O(P_1 \times \gamma_k^n)$ and is $O^*(\gamma_k^n)$, where $\gamma_k < 2$ is the largest real root of the equation $x^{k+3} - 2x^{k+2} + 1 = 0$.*

Proof. Firstly, as the search tree $\mathcal{T}$ is a full binary tree, the total number of nodes in the search tree is at most twice the number of its leaf nodes. Secondly, it is easy to see that the time complexity of each node is P_1. Thus, the theorem holds. □

The existing best time complexity for the problem of maximum k-defective clique computation is achieved by the algorithm MADEC$^+$ [11], which is $O^*(\sigma_k^n)$ where σ_k is the largest real root of $x^{2k+3} - 2x^{2k+2} + 1 = 0$. It is easy to see that $\sigma_k = \gamma_{2k}$. Thus, the time complexity of MADEC$^+$ is $O^*(\gamma_{2k}^n)$ and is higher than our time complexity considering that $\gamma_k < \gamma_{2k}$. Our algorithm has two main features that enable the improved time complexity. Firstly, after exhaustively applying our reduction rules **RR1** and **RR2**, every vertex in $V(g) \setminus S$ will have at least two non-neighbors as stated in Lemma 3.3; as a result, we can prove **Fact 3** above. If we do not apply **RR2**, then we will only be able to bound q by $2k + 1$ for **Fact 3** and get the same time complexity as [11]. For example, consider the graph in Figure 4 again and suppose $k = 2$ and $I_0.S = \{v_1\}$. The branching vertex selected for I_0 could be $b_0 = v_2$, and we have $I_1.S = \{v_1, v_2\}$. The branching vertex selected for I_1 could be $b_1 = v_4$ and we get $I_2.S = \{v_1, v_2, v_4\}$. Similarly, we could have $b_2 = v_3$, $I_3.S = \{v_1, \ldots, v_4\}$, $b_3 = v_5$, $I_4.S = \{v_1, \ldots, v_5\}$, $b_4 = v_6$ and $I_5.S = \{v_1, \ldots, v_6\}$. Only in instance I_5, the reduction rules will finally have any effect (e.g., remove vertices $\{v_8, v_9\}$); thus $q = 5$. The second feature of our algorithm is our new branching rule. We remark that after incorporating our reduction rules **RR1** and **RR2** into [11], its time complexity could be improved to roughly $O^*(\gamma_{3k/2}^n)$. However, this is still higher than our time complexity. Thus, our branching rule is better than that of MADEC$^+$ [11].

It is interesting to observe that the state-of-the-art time complexity for maximum k-plex computation is similar to $O^*(\gamma_k^n)$; specifically, it is $O^*(\gamma_{k-1}^n)$ [13, 50]. This is because an inequality similar to Equation (1) is also proved and utilized in [13, 50]. But, we remark that our techniques and arguments to obtain Equation (1) are different from that of [13, 50] due to different problem natures.

Algorithm 2: kDC(G, k)

```
1 $C^* \leftarrow$ heuristically compute a large $k$-defective clique of $G$;
2 $g \leftarrow$ apply reduction rules to reduce $G$;
3 Branch&Bound$(g, \emptyset)$;
4 **return** $C^*$;
```

3.1.3 *Incorporating Other Practical Techniques.*

Algorithm 1 is used for illustrating the bare minimum needed to achieve our time complexity of $O^*(\gamma_k^n)$, and its practical performance would not be satisfactory. Thus, we propose to further incorporate other practical techniques, such as preprocessing, upper bound-based pruning and more reduction rules, into our algorithm. The pseudocode of our practically improved algorithm kDC is given in Algorithm 2. In kDC, we first heuristically compute a large k-defective clique of G (Line 1), whose details will be given in Section 3.3. Let C^* be the heuristically computed k-defective clique. We then use $|C^*|$ to reduce G by removing unpromising vertices and edges (Line 2); the details will be given in Section 3.2.3. After that, we go into backtracking by invoking the procedure Branch&Bound.

Branch&Bound is similar to Branch&Bound-t in Algorithm 1, but with the following additions. Firstly, besides **RR1** and **RR2**, we also apply other reduction rules at Line 4 of Algorithm 1; these reduction rules will be presented in Section 3.2.2. Secondly, before picking a branching vertex at Line 6 of Algorithm 1, we also compute an upper bound of the maximum k-defective clique in the instance (g', S'), and prune the entire instance if the computed upper bound is no larger than $|C^*|$; this implies that no k-defective clique in the instance (g', S') is of size larger than C^*. Details of the upper bound computation will be presented in Section 3.2.1.

Let P_2 be the time complexity of running Lines 1–2 of Algorithm 2. The time complexity of kDC is $O(P_2 + P_1 \times \gamma_k^n)$ and is also $O^*(\gamma_k^n)$. That is, applying these additional techniques does not affect the exponential part of our time complexity in Theorem 3.5.

3.2 Upper Bounds and Reduction Rules

3.2.1 Upper Bounds. For the upper bound-based pruning, we propose an improved graph coloring-based upper bound. Before that, we first present the existing graph coloring-based upper bound proposed in [11], where the graph coloring is mainly used to partition the vertices into independent sets. Specifically, a coloring of a graph is assigning each vertex a color such that for every edge in the graph, its two end-points have different colors. Given an instance (g, S) and a coloring of $V(g) \setminus S$ with c distinct colors $\{1, 2, \ldots, c\}$, let $\pi_1, \pi_2, \ldots, \pi_c$ be the partitioning of $V(g) \setminus S$ based on their colors; that is, each π_i consists of all vertices with color i and thus is an independent set. The upper bound in [11] is

$$|S| + \sum_{i=1}^{c} \min\left(\left\lfloor \frac{1+\sqrt{8k+1}}{2} \right\rfloor, |\pi_i|\right) \tag{2}$$

This is based on the observation that an independent set with more than $\lfloor \frac{1+\sqrt{8k+1}}{2} \rfloor$ vertices will miss more than k edges and thus cannot be all contained in the same k-defective clique. However, the upper bound computed by Equation (2) has two deficiencies.

- It considers the partitions $\pi_1, \ldots, \pi_c$ independently, and thus would include much more vertices than necessary for computing the upper bound. For example, suppose $|\pi_i| \geq \lfloor \frac{1+\sqrt{8k+1}}{2} \rfloor$, $\forall 1 \leq i \leq c$, then the upper bound becomes $|S| + c \cdot \lfloor \frac{1+\sqrt{8k+1}}{2} \rfloor$. But obviously $|S| + c + k$ is a much smaller upper bound (e.g., when c is large); this is because adding any s_i vertices of π_i to S will introduce at least $s_i - 1$ non-edges.
- It does not consider the non-edges in S, and the non-edges between S and $V(g) \setminus S$.

As a result, the upper bound computed by Equation (2) is not tight.

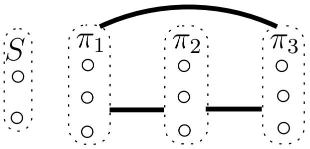


Figure 5: Running example for graph coloring-based upper bound computation (the graph consists of 11 vertices and 27 edges where (π_1, π_2, π_3) forms a 3-partite clique, i.e., all edges are between a vertex of π_i and a vertex of π_j for $i \neq j$)

Example 3.6. Consider the graph g in Figure 5 and the partial solution S consisting of two isolated vertices (i.e., without any adjacent edges). Besides S, the other part of g is a 3-partite clique with vertex sets π_1, π_2 and π_3; note that, there is no edge between S and $V(g) \setminus S$. Thus, a graph coloring of $V(g) \setminus S$ would assign all vertices of π_i with color i, for $1 \leq i \leq 3$. Suppose $k = 3$, then $\lfloor \frac{1+\sqrt{8k+1}}{2} \rfloor = 3$. As $|\pi_1| = |\pi_2| = |\pi_3| = 3$, the graph coloring-based upper bound computed by Equation (2) is $2 + 3 \times 3 = 11$. However, it is easy to observe that the maximum k-defective clique in the instance (g, S) is of size only 3, as we can only add one more vertex without violating the k-defective clique definition.

In this paper, we still utilize the graph coloring-based partitioning $\pi_1, \ldots, \pi_c$ for computing the upper bound, but we compute a much tighter (i.e., smaller) upper bound than Equation (2) by resolving the above two deficiencies, as follows.

UB1 (improved coloring-based upper bound). For each partition π_i, we first sort its vertices into non-decreasing order regarding $|\overline{N}_S(\cdot)|$ and then define the weight of the j-th vertex in the sorted order, denoted v_{i_j}, to be $\mathrm{w}(v_{i_j}) = |\overline{N}_S(v_{i_j})| + j - 1$, where the index j starts from 1. Finally, let $v_1, v_2, \ldots,$ be an ordering of $V(g) \setminus S$ in non-decreasing order regarding their weights $\mathrm{w}(\cdot)$. The maximum k-defective clique in the instance (g, S) is of size at most $|S|$ plus the largest i such that $\sum_{j=1}^{i} \mathrm{w}(v_j) \leq k - |\overline{E}(S)|$.

The general idea is that (1) adding any $s_i \geq 0$ vertices of π_i to S will introduce at least $\sum_{j=1}^{s_i} \mathrm{w}(v_{i_j}) \geq \frac{s_i(s_i-1)}{2}$ non-edges, and (2) $(s_1, s_2, \ldots, s_c)$ is determined greedily and interdependently. It can be verified that the upper bound computed by **UB1** is at most (and can be much smaller than) $|S| + c + k - |\overline{E}(S)|$, and is also no larger than that computed by Equation (2).

Proof of UB1. For any k-defective clique C in the instance (g, S) that contains s_i vertices of π_i for each $1 \leq i \leq c$, the number of missing edges in C is

$$\begin{aligned} |\overline{E}(C)| &\geq |\overline{E}(S)| + \sum_{i=1}^{c} \left(\frac{s_i(s_i-1)}{2} + \sum_{j=1}^{s_i} |\overline{N}_S(v_{i_j})| \right) \\ &= |\overline{E}(S)| + \sum_{i=1}^{c} \sum_{j=1}^{s_i} \left(|\overline{N}_S(v_{i_j})| + j - 1 \right) \\ &= |\overline{E}(S)| + \sum_{i=1}^{c} \sum_{j=1}^{s_i} \mathrm{w}(v_{i_j}) \\ &\geq |\overline{E}(S)| + \sum_{j=1}^{s_1+\cdots+s_c} \mathrm{w}(v_j) \end{aligned}$$

The first inequality follows from the fact that adding s_i vertices S_i of π_i to S will introduce

- at least $\frac{s_i(s_i-1)}{2}$ non-edges between vertices of S_i (since S_i is an independent set), and
- at least $\sum_{j=1}^{s_i} |\overline{N}_S(v_{i_j})|$ non-edges between S and S_i (since the vertices of π_i are ordered such that $|\overline{N}_S(v_{i_1})| \leq |\overline{N}_S(v_{i_2})| \leq |\overline{N}_S(v_{i_3})| \leq \cdots$).

The second inequality follows from the fact that the vertices of $V(g) \setminus S$ are ordered such that $\mathrm{w}(v_1) \leq \mathrm{w}(v_2) \leq \mathrm{w}(v_3) \leq \cdots$. Consequently, **UB1** follows from the fact that $|\overline{E}(C)| \leq k$ as C is a k-defective clique. □

Example 3.7. Continue Example 3.6. Now we show how **UB1** computes the upper bound. As $|\overline{N}_S(v)| = 2$ for each $v \in V(g) \setminus S$, the weights of the vertices in π_i for $1 \leq i \leq 3$ are all $\{2, 3, 4\}$. Thus, the weights of all vertices of $V(g) \setminus S$ are $\{2, 2, 2, 3, 3, 3, 4, 4, 4\}$. As $|\overline{E}(S)| = 1$, the upper bound computed by **UB1** is $2 + 1 = 3$, which is significantly tighter than that computed in Example 3.6.

In addition, we also adopt the following two upper bounds from the literature.

UB2. The maximum k-defective clique in the instance (g, S) is of size at most $\min_{u \in S} d_g(u) + 1 + k$ [11].

UB3. Given an instance (g, S), let $v_1, v_2, \ldots$ be an ordering of $V(g) \setminus S$ in non-decreasing order regarding their numbers of non-neighbors in S, i.e.$|\overline{N}_S(\cdot)|$. The maximum k-defective clique in the instance (g, S) is of size at most $|S|$ plus the largest i such that $\sum_{j=1}^{i} |\overline{N}_S(v_j)| \leq k - |\overline{E}(S)|$ [16].

3.2.2 Reduction Rules. Besides the two reduction rules **RR1** and **RR2** presented in Section 3.1, we further propose two new reduction rules based on the size of the currently found best solution (i.e., currently found largest k-defective clique). ***Let*** lb ***be the size of the currently found best solution***, then we will not be interested in any solution of size $\leq lb$. We first present and prove the reduction rule **RR3** that is derived from the upper bound **UB3**.

RR3 (degree-sequence-based reduction rule). Given an instance (g, S), let $v_1, v_2, \ldots$ be an ordering of $V(g) \setminus S$ in non-decreasing order regarding their numbers of non-neighbors in S, i.e., $|\overline{N}_S(\cdot)|$. For a vertex v_i with $i > lb - |S|$ and $|\overline{N}_S(v_i)| > k - |\overline{E}(S)| - \sum_{j=1}^{lb-|S|} |\overline{N}_S(v_j)|$, we can remove v_i from g.

Proof of RR3. Consider the instance $(g, S \cup v_i)$ which is obtained from (g, S) by adding $v_i \in V(g) \setminus S$ to S, and denote $S \cup v_i$ by S'. Let $v'_1, v'_2, \ldots$ be an ordering of $V(g) \setminus S'$ in non-decreasing order regarding their numbers of non-neighbors in S'. **UB3** states that the maximum k-defective clique in the instance (g, S') is of size at most $|S'|$ plus the largest i' such that $\sum_{j=1}^{i'} |\overline{N}_{S'}(v'_j)| \leq k - |\overline{E}(S')|$. Note that, we have

$$\begin{aligned}\sum_{j=1}^{lb-|S|} |\overline{N}_{S'}(v'_j)| &\geq \sum_{j=1}^{lb-|S|} |\overline{N}_S(v_j)| \\ &> k - |\overline{E}(S)| - |\overline{N}_S(v_i)| = k - |\overline{E}(S')|\end{aligned}$$

where the first inequality follows from the fact that $|\overline{N}_{S'}(v)| \geq |\overline{N}_S(v)|$ holds for every $v \in V(g) \setminus S'$, and the second inequality follows from the statement of the reduction rule **RR3**. Consequently, the maximum k-defective clique in the instance (g, S') is of size strictly less than $|S'| + (lb - |S|) = lb + 1$. That is, every k-defective clique that is in the instance (g, S) and contains v_i is of size at most lb, and thus we can remove v_i from g. □

Then, we propose the reduction rule **RR4**.

RR4 (second-order reduction rule). Given an instance (g, S) with $S \neq \emptyset$, for any vertex $u \in S$ and $v \in V(g) \setminus S$, let $S' = S \cup v$, $cn(u, v)$ be the number of common neighbors of u and v in $\overline{S'} = V(g) \setminus S'$, $cnon(u, v)$ be the number of common non-neighbors of u and v in $\overline{S'}$, $xn(u, v)$ be the number of vertices that are exclusive neighbors of either u or v in $\overline{S'}$; specifically, $xn(u, v) = |N_{\overline{S'}}(u) \setminus N_{\overline{S'}}(v)| + |N_{\overline{S'}}(v) \setminus N_{\overline{S'}}(u)|$.
If $|S'| + cn(u, v) + \min\big(k - |\overline{E}(S')|, xn(u, v)\big) + \min\big(cnon(u, v), \max(0, \lfloor \frac{k - |\overline{E}(S')| - xn(u,v)}{2} \rfloor)\big) \leq lb$, then we can remove v from g.

Proof of RR4. Let's consider the instance (g, S'). It is easy to verify that $cn(u, v)$, $cnon(u, v)$ and $xn(u, v)$ represent disjoint subsets of vertices of $V(g) \setminus S'$ and $cn(u, v) + cnon(u, v) + xn(u, v) = |V(g) \setminus S'|$. We consider two cases depending on whether $k - |\overline{E}(S')| > xn(u, v)$. Firstly, if $k - |\overline{E}(S')| \leq xn(u, v)$, then the maximum k-defective clique in the instance (g, S') will be of size at most $|S'| + cn(u, v) + (k - |\overline{E}(S')|)$, since (1) adding any exclusive neighbor of u or v to S' will introduce at least one non-edge and (2) adding any common non-neighbor of u and v to S' will introduce at least two non-edges. Similarly, if $k - |\overline{E}(S')| > xn(u, v)$, then the maximum k-defective clique in the instance (g, S') will be of size at most $|S'| + cn(u, v) + xn(u, v) + \min(cnon(u, v), \lfloor \frac{k - |\overline{E}(S')| - xn(u,v)}{2} \rfloor)$. In summary, the maximum k-defective clique in the instance (g, S') is of size at most $|S'| + cn(u, v) + \min\big(k - |\overline{E}(S')|, xn(u, v)\big) + \min\big(cnon(u, v), \max(0, \lfloor \frac{k - |\overline{E}(S')| - xn(u,v)}{2} \rfloor)\big)$ and **RR4** is correct. □

From the proofs of **RR3** and **RR4**, it can be observed that the reduction rules are actually designed based on upper bounds; for example, **RR3** is based on **UB3**. The general idea is that, given an instance (g, S) and a vertex $v \in V(g) \setminus S$, if an upper bound of the instance $(g, S \cup v)$ is at most lb, then we can remove v from g. It is easy to see that an alternative strategy is to directly generate the instance $(g, S \cup v)$ which will then be pruned by the upper bounds; this will have the same pruning effects as the reduction rules. The advantage of using reduction rules to remove v from g is that the reduction rules can be applied more efficiently by computation sharing; in particular, applying the reduction rules for all vertices of $V(g) \setminus S$ can be conducted in linear time in total (see Section 3.2.3), while generating all the sub-instances and then pruning by the upper bounds would take quadratic time. On the other hand, it is also worth mentioning that an upper bound could be designed based on **RR4**; we do not use it in this paper since computing this upper bound is time-consuming.

In addition, we also utilize the following two reduction rules from the literature.

RR5. Given an instance (g, S), for any vertex $v \in V(g) \setminus S$ whose degree is less than $lb - k$, we can remove v from g [11].

RR6. Given a graph G, for any edge $(u, v) \in E(G)$ whose number of common neighbors in G is less than $lb - k - 1$, we can remove the edge (u, v) from G [16].

3.2.3 Time Complexity Analysis. Now, we analyze the time complexity of all the upper bounds and reduction rules. Firstly, for the upper bounds **UB1**–**UB3**, it is easy to see that **UB2** and **UB3** can be computed in time linear to the number of edges (i.e., $O(m)$); note that, sorting vertices in **UB3** can be conducted in linear time by counting sort [12]. For **UB1**, we use the widely adopted greedy approach to assign colors to vertices [8, 43]; that is, colors are assigned to vertices in the reverse order of the degeneracy ordering (see Definition 2.3 for the definition of degeneracy ordering), and a vertex is assigned the smallest color that has not been taken by its neighbors. Consequently, **UB1** can be computed in $O(m)$ time.

Secondly, for the reduction rules **RR1**–**RR5**, it is easy to see that **RR1**, **RR2** and **RR3** can be exhaustively applied until convergence (i.e., until the instance can no longer be reduced by these reduction rules) in linear time. For **RR4**, we do not apply it exhaustively for the sake of efficiency. Instead, we let u be the vertex most recently added to S, and loop through each vertex $v \in V(g) \setminus S$ only once. Observing that $cn(u, v) = |N_{\overline{S'}}(u) \cap N(v)|$, $cnon(u, v) = |\overline{N}_{\overline{S'}}(u)| - |\overline{N}_{\overline{S'}}(u) \cap N(v)|$, and $xn(u, v) = |V(g) \setminus S'| - cn(u, v) - cnon(u, v)$, applying **RR4** for u and v can be conducted in $O(d_g(v))$ time by marking $N_{\overline{S'}}(u)$ and $\overline{N}_{\overline{S'}}(u)$ in a preprocessing step. Consequently, applying **RR4** once for all vertices $v \in V(g) \setminus S$ takes $O(m)$ time in total. For the reduction rule **RR5**, it actually reduces the graph g to its $(lb - k)$-core (see Definition 2.4), i.e., a vertex is removed from

g if its degree in g is smaller than $lb - k$; this can be conducted in $O(m)$ time [10, 28]. Note that, if a vertex of S is removed during the process, then the instance (g, S) is pruned (based on **UB2**).

Thirdly, for the reduction rule **RR6**, we only apply it in the preprocessing (i.e., Line 2 of Algorithm 2) as it has a higher time complexity than other reduction rules. Specifically, exhaustively applying **RR6** actually reduces the input graph G to its $(lb - k + 1)$-truss (see Definition 2.5), i.e., an edge (u, v) is removed from G if the number of common neighbors of its two end-points in G is smaller than $lb - k - 1$; this can be conducted in $O(\delta(G) \times m)$ time [46], where $\delta(G)$ is the degeneracy of G and is at most $\sqrt{m}$.

In summary, we apply reduction rules **RR1**–**RR5** and upper bounds **UB1**–**UB3** in Branch&Bound; thus, P_1 in Theorem 3.5 is $O(m)$. For Line 2 of Algorithm 2, we first exhaustively apply **RR5**, and then **RR6**. Thus, Line 2 of Algorithm 2 takes $O(\delta(G) \times m)$ time.

3.3 Compute a Large Initial Solution

In this subsection, we discuss how to efficiently compute a large initial k-defective clique at Line 1 of Algorithm 2. Firstly, we can heuristically compute a k-defective clique in $O(m)$ time based on the degeneracy ordering, i.e., the longest suffix of the degeneracy ordering that is a k-defective clique. The pseudocode is shown in Algorithm 3, denoted Degen.

```
Algorithm 3: Degen(G, k)
Input: A graph $G$ and an integer $k$
Output: A large $k$-defective clique in $G$
Compute a degeneracy ordering for the vertices of $G$;
$C \leftarrow$ the longest suffix of the degeneracy ordering that is a $k$-defective clique;
return $C$;
```

As discussed at the end of Section 3.2.3, Line 2 of Algorithm 2 takes $O(\delta(G) \times m)$ time; this is higher than the time complexity of Degen. Thus, it makes sense to spend a little more time at Line 1 of Algorithm 2 aiming to compute a larger initial solution. Motivated by this, besides heuristically computing a degeneracy ordering-based solution in the input graph G, we also extract n subgraphs from G — one subgraph for each vertex of G — and heuristically compute a degeneracy ordering-based solution in each of the subgraphs; the largest one among these $n + 1$ solutions is then kept as the initial solution. To bound the time complexity by $O(\delta(G) \times m)$, we extract the subgraphs based on a degeneracy ordering of G. Specifically, let $(v_1, v_2, \ldots, v_n)$ be a degeneracy ordering of G, the subgraph extracted for v_i then is the subgraph of G induced by the set of higher ranked neighbors of u regarding the degeneracy ordering, i.e., $N(v_i) \cap \{v_{i+1}, \ldots, v_n\}$. The pseudocode is shown in Algorithm 4, denoted Degen-opt. As the subgraph extracted for v_i will have at most $\min(d(v_i), \delta(G))$ vertices, the time complexity of Algorithm 4 is bounded by

$$\sum_{i=1}^{n} \min(d(v_i), \delta(G))^2 \leq \sum_{i=1}^{n} d(v_i) \times \delta(G) = 2 \times \delta(G) \times m$$

Consequently, by invoking Algorithm 4 at Line 1 of Algorithm 2, P_2 in Section 3.1.3 is $O(\delta(G) \times m)$, and the time complexity of kDC is $O(\delta(G) \times m + m \times \gamma_k^n)$.

Example 3.8. Consider the graph in Figure 6, a degeneracy ordering is $(v_1, v_2, v_5, v_3, v_4, v_6, v_7)$. Suppose $k = 1$, the longest suffix that is a k-defective clique is $\{v_4, v_6, v_7\}$ of size 3; thus, Degen finds an initial solution of size 3. Now, let's consider v_1, its set of higher ranked neighbors is $N^+(v_1) = \{v_2, v_3, v_4\}$. It is easy to see that the subgraph induced by $N^+(v_1)$ is a k-defective clique. Thus, Degen-opt reports the initial solution $\{v_1, v_2, v_3, v_4\}$ of size 4.

```
Algorithm 4: Degen-opt(G, k)
Input: A graph $G$ and an integer $k$
Output: A large $k$-defective clique in $G$
$C \leftarrow$ Degen$(G, k)$;
Compute a degeneracy ordering for the vertices of $G$;
for each vertex $u \in V(G)$ do
    $N^+(u) \leftarrow$ the set of higher ranked neighbors of $u$ in $G$, according to the degeneracy ordering;
    $g \leftarrow$ the subgraph of $G$ induced by $N^+(u)$;
    $C' \leftarrow$ Degen$(g, k)$;
    if $|C' \cup u| > |C|$ then $C \leftarrow C' \cup u$;
return $C$;
```

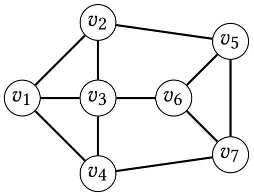


Figure 6: Running example for computing an initial solution

4 EXPERIMENTS

We have shown in Section 3.1 that our algorithm kDC achieves a better time complexity than the existing algorithms for the problem of maximum k-defective clique computation. In this section, we show empirically that kDC also performs better than the existing algorithms in practice. Specifically, we evaluate kDC against the following existing algorithms.

- KDBB: the existing algorithm with the state-of-the-art practical performance proposed in [16].
- MADEC$^+_\mathrm{p}$: the existing algorithm with the state-of-the-art time complexity proposed in [11].

In addition, we also evaluate the following variants of our algorithm kDC to test the effectiveness of the different components of kDC.

- kDC/UB1: kDC without the upper bound **UB1**.
- kDC/RR3&4: kDC without the reduction rules **RR3** and **RR4**.
- kDC-Degen: kDC with the initial solution (i.e., Line 1 of Algorithm 2) computed by Degen and without applying the reduction rule **RR6** at Line 2 of Algorithm 2.

All our algorithms are implemented in C++ and compiled with -O3 optimization. [2] All experiments are run in the single-thread mode on a machine with an Intel Core i7-8700 CPU and 64GB main memory and running Ubuntu 18.04.

Datasets. Same as [16], we run the algorithms on the following three graph collections.

[2] The source code of kDC is released at https://lijunchang.github.io/Maximum-kDC/

Table 2: Number of solved instances by the algorithms kDC, KDBB and $\mathsf{MADEC}^+_\mathsf{p}$ with a time limit of 3 hours (best performers are highlighted in bold)

	Real-world graphs			Facebook graphs			DIMACS10&SNAP		
	kDC	KDBB	$\mathsf{MADEC}^+_\mathsf{p}$	kDC	KDBB	$\mathsf{MADEC}^+_\mathsf{p}$	kDC	KDBB	$\mathsf{MADEC}^+_\mathsf{p}$
$k = 1$	**133**	117	115	**114**	110	110	**37**	36	36
$k = 3$	**130**	107	94	**114**	110	104	**37**	35	31
$k = 5$	**127**	104	81	**114**	108	78	**37**	34	28
$k = 10$	**119**	85	36	**111**	109	9	**36**	30	15
$k = 15$	**110**	68	26	101	**103**	0	**29**	25	10
$k = 20$	**104**	56	20	**88**	80	0	**27**	22	6

- The **real-world graphs** collection [3] contains 139 real-world graphs from the Network Data Repository with up to 5.87×10^7 vertices and 1.06×10^8 undirected edges.
- The **Facebook graphs** collection [4] contains 114 Facebook social networks from the Network Data Repository with up to 5.92×10^7 vertices and 9.25×10^7 undirected edges.
- The **DIMACS10&SNAP graphs** collection contains 37 graphs with up to 1.04×10^6 vertices and 6.89×10^6 undirected edges. Among the 37 graphs, 27 are from DIMACS10 [5] and 10 are from SNAP [6].

Note that, the graphs included in these three collections are the same ones tested in [16].

Metric. We record the total *processing time* of running an algorithm on a graph instance for a specific k. The recorded processing time is the total CPU time excluding the I/O time of loading the graph instance from disk to main memory. Same as [16], we choose k from $\{1, 3, 5, 10, 15, 20\}$ and set a time limit of 3 hours for each testing.

4.1 Against the Existing Algorithms

In this subsection, we evaluate our algorithm kDC against the existing algorithms KDBB and $\mathsf{MADEC}^+_\mathsf{p}$, regarding the efficiency. The results on the number of solved instances with a time limit of 3 hours are shown in Table 2, which are also partially illustrated in Figures 7 and 8. $\mathsf{MADEC}^+_\mathsf{p}$ is an improved version, by the authors of KDBB [16], of the MADEC^+ algorithm proposed in [11]. As *the authors of [16] are not able to provide the code of algorithms* KDBB *and* $\mathsf{MADEC}^+_\mathsf{p}$, their numbers reported in Table 2 are obtained from the original paper [16]; note that [16] tests exactly the same sets of graphs for the three graph collections and also has the time limit of 3 hours. From Table 2, we can see that KDBB significantly outperforms $\mathsf{MADEC}^+_\mathsf{p}$ (especially for $k \geq 5$), and our algorithm kDC further outperforms KDBB with the only exception of $k = 15$ on the Facebook graphs collection. We also would like to highlight two other observations that can be observed from Figures 7 and 8. Firstly, on the real-world graphs collection, kDC with a time limit of 3 *seconds* solves even more instances than KDBB with a time limit of 3 *hours*. Secondly, on the Facebook graphs collection, kDC solves all 114 instances with time limits of 125, 393 and 1353 seconds, respectively, for $k = 1$, 3 and 5. This demonstrates the practical superiority of kDC over the existing algorithms.

To dive into a more detailed performance gain of kDC over KDBB, we report the actual processing time of kDC and KDBB on the subset of Facebook graphs that have more than $15,000$ vertices; there are 41 such graphs. We report the results for $k = 1$, 3, 5, and 10 in Table 3, as KDBB only gave results for such k values in [16]; for now, please ignore the columns regarding algorithms kDC/RR3&4, kDC/UB1, and kDC-Degen. The number of vertices and edges in these graphs are also illustrated in Table 3. Note that, an '–' for KDBB indicates that the result is not available (specifically, [16] didn't report the results on the four graphs A-anon, B-anon, konect, uci-uni), while an '–' for our algorithms indicates that the processing time is longer than the 3-hour limit. From Table 3, we can observe that kDC consistently and significantly runs faster than KDBB across all these testings. In particular, kDC on average is 1552, 1754, 1636 and 820 times faster than KDBB for $k = 1$, 3, 5 and 10, respectively. This further demonstrates the superior performance of kDC over the existing fastest algorithm KDBB.

We also would like to discuss the performance of our algorithm kDC on some large-scale graphs. Firstly, for the two graphs, konect and uci-uni, in the Facebook graphs collection that have 58M vertices and 92M undirected edges (here M means $\times 10^6$), kDC is able to find the maximum k-defective clique within the time limit for $k \leq 5$ and times out for larger k values; please refer to Table 3 for the results. Secondly, for the soc-orkut graph in real-world graphs collection that have 3M vertices and 106M undirected edges, kDC is able to find the maximum k-defective clique within the time limit for $k \leq 3$ and times out for larger k values. Thirdly, we also tested kDC on the webbase-2001 graph (downloaded from https://law.di.unimi.it/datasets.php) that has 116M vertices and 855M undirected edges. kDC is able to find the maximum k-defective clique for all the tested k values within 30 seconds. It will be our future work to further improve the number of solved instances for the different k values.

4.2 Ablation Studies

Now, we conduct ablation studies for our proposed techniques. Firstly, we compare kDC with kDC/RR3&4 which is kDC without applying our two reduction rules **RR3** and **RR4** that are described in Section 3.2.2. The results for $k = 1$, 3, 5, 10, 15, and 20 on the real-world graphs collection are shown in Figure 7, and that on the Facebook graphs collection are shown in Figure 8; specifically, we vary the time limit and report the number of graph instances that are solved by an algorithm within a specific time limit. We can see that kDC consistently outperforms kDC/RR3&4, and the improvement is more evident when k becomes large; for example, for $k = 20$ and with a time limit of 3 hours, kDC solves 13 and 15 more instances than KDBB on the real-world graphs collection and the Facebook graphs collection, respectively. This demonstrates that our new reduction rules **RR3** and **RR4** are effective in improving the efficiency of maximum k-defective computation. However, we also observe that kDC and kDC/RR3&4 perform similarly for $k \leq 10$ on the Facebook graphs collection; specifically, the processing time of kDC/RR3&4 on 41 of the Facebook graphs for $k = 3$ and $k = 10$ are also listed in Table 3. One of the reasons is that the upper bound **UB1**, which is used in both kDC and kDC/RR3&4, is very effective for these testings, and as a result **RR3** and **RR4** do not prune many additional vertices. To verify that, we also implement a version of kDC without **UB1**, **RR3** and **RR4**, denoted kDC/UB1&RR3&4, and compare it with kDC/UB1. Our results show that kDC/UB1

[3] http://lcs.ios.ac.cn/~caisw/Resource/realworld%20graphs.tar.gz

[4] https://networkrepository.com/socfb.php

[5] https://www.cc.gatech.edu/dimacs10/downloads.shtml

[6] http://snap.stanford.edu/data/

Table 3: Processing time (in seconds) of kDC, kDC/RR3&4, kDC/UB1, kDC-Degen **and** KDBB **on the** 41 **Facebook graphs with more than** 15, 000 **vertices; the results of** kDC/RR3&4, kDC/UB1 **and** kDC-Degen **for** $k = 1$ **and** $k = 5$ **are omitted due to space limitations. Best performers are highlighted in bold; if a running time is slow than the fastest running time by less than 10%, we also consider it to be best.** n **is the number of vertices and** m **is the number of edges in the graph.**

			$k = 1$		$k = 3$					$k = 5$		$k = 10$				
	n	m	kDC	KDBB	kDC	kDC/RR3&4	kDC/UB1	kDC-Degen	KDBB	kDC	KDBB	kDC	kDC/RR3&4	kDC/UB1	kDC-Degen	KDBB
A-anon	3M	23M	**5.0**	-	**5.2**	**5.1**	29	6803	-	**5.7**	-	**73**	412	6540	-	-
Auburn71	18K	973K	**1.4**	432	1.9	**1.6**	384	55	536	**8.6**	639	**956**	**905**	-	-	1195
B-anon	2M	20M	**7.9**	-	**8.4**	**8.1**	57	-	-	**9.2**	-	**44**	56	7858	-	-
Berkeley13	22K	852K	**0.18**	425	**0.18**	**0.18**	0.42	6.5	452	**0.19**	506	**0.34**	0.39	61	55	630
BU10	19K	637K	**0.09**	252	**0.15**	**0.15**	1.1	6.5	290	**0.39**	332	**4.0**	5.1	16	35	370
Cornell5	18K	790K	**1.1**	393	**2.1**	**2.0**	194	249	922	**2.6**	1265	**17**	**16**	8670	-	2636
FSU53	27K	1M	**0.35**	209	0.40	**0.23**	-	80	610	**2.8**	828	248	**221**	-	8351	1400
Harvard1	15K	824K	**0.76**	347	**0.82**	**0.85**	91	284	421	**0.94**	517	**11**	**10**	3680	-	1354
Indiana	29K	1M	**0.46**	1142	**0.46**	**0.48**	21	95	1138	**0.53**	1261	**19**	**20**	1975	3710	1421
Indiana69	29K	1M	**0.46**	1134	**0.46**	**0.48**	21	97	1072	**0.54**	1186	**19**	**20**	1964	3706	1321
konect	59M	92M	**8.1**	-	**8.2**	**7.9**	9.7	178	-	**9.4**	-	-	-	-	-	-
Maryland58	20K	744K	**0.10**	150	**0.12**	**0.11**	0.18	4.1	162	**0.12**	185	**0.60**	0.82	2.7	8.2	239
Michigan23	30K	1M	**0.63**	833	**0.66**	**0.63**	0.88	1556	1072	**0.67**	971	**2.2**	2.9	215	-	1384
MSU24	32K	1M	**0.35**	493	**0.35**	**0.33**	0.40	92	576	**0.34**	666	**0.47**	**0.50**	1.5	10227	879
MU78	15K	649K	**0.10**	182	**0.13**	**0.12**	0.53	1.1	200	**0.32**	215	**67**	**68**	393	203	306
NYU9	21K	715K	**0.09**	349	**0.09**	**0.09**	0.13	17	399	**0.09**	396	**0.12**	0.13	0.17	26	466
Oklahoma97	17K	892K	**0.78**	383	**0.96**	**0.89**	-	1162	2048	**5.1**	3938	379	**334**	-	10533	6926
OR	63K	816K	**0.16**	356	**0.30**	0.34	21	9.9	456	**1.0**	587	**55**	**55**	885	258	1486
Penn94	41K	1M	**0.23**	1139	**0.23**	**0.22**	0.25	8.2	1557	**0.23**	1820	**0.29**	**0.32**	0.35	20	1972
Rutgers89	24K	784K	**0.08**	219	0.07	0.07	0.09	**0.04**	276	**0.08**	279	**0.20**	**0.22**	1.4	7.1	386
Tennessee95	16K	770K	**0.54**	246	**0.56**	**0.53**	2.7	16	361	**0.52**	424	**1.8**	**1.7**	62	884	554
Texas80	31K	1M	**0.56**	342	**0.66**	**0.62**	4.6	52	423	**0.73**	534	80	**70**	1136	2603	753
Texas84	36K	1M	**6.2**	1490	13	**11**	6503	5555	1674	**69**	2769	1321	**1134**	-	-	10253
UC33	16K	522K	**0.07**	156	**0.07**	**0.06**	1.6	1.3	171	**0.07**	181	**0.14**	**0.15**	148	3.2	263
uci-uni	58M	92M	**13**	-	**13**	**12**	14	206	-	**14**	-	-	-	-	-	-
UCLA	20K	747K	**0.09**	190	0.09	0.09	0.11	**0.04**	206	**0.09**	237	**0.14**	**0.15**	0.17	4.9	290
UCLA26	20K	747K	**0.09**	184	0.09	0.09	0.12	**0.04**	207	**0.09**	215	**0.14**	**0.15**	0.19	4.9	288
UConn	17K	604K	**0.06**	109	0.06	0.05	0.06	**0.04**	126	**0.06**	169	**0.13**	0.16	0.22	2.5	194
UConn91	17K	604K	**0.06**	105	0.06	0.05	0.07	**0.04**	123	**0.06**	173	**0.13**	0.16	0.22	2.5	208
UF	35K	1M	**0.58**	793	**0.58**	0.84	-	282	1332	**0.74**	1602	**27**	**29**	-	8777	2579
UF21	35K	1M	**0.57**	787	**0.58**	0.84	-	281	1297	**0.74**	1542	**27**	**29**	-	8767	2571
UGA50	24K	1M	**5.7**	724	43	**37**	-	4895	1467	**165**	2459	3318	**2856**	-	-	6794
UIllinois	30K	1M	**0.68**	486	**0.69**	**0.65**	2.9	93	644	**0.65**	806	**3.6**	**3.5**	342	8237	1245
UIllinois20	30K	1M	**0.68**	486	**0.68**	**0.65**	2.9	93	610	**0.66**	784	**3.6**	**3.5**	341	8195	1217
UMass92	16K	519K	**0.15**	226	**0.15**	**0.15**	0.19	25	245	**0.17**	265	**0.30**	0.37	0.58	82	318
UNC28	18K	766K	**0.46**	236	**0.47**	**0.45**	0.82	54	287	**0.46**	336	**2.1**	**2.1**	15	7278	380
USC35	17K	801K	**0.31**	232	**0.31**	**0.30**	0.60	390	267	**0.31**	334	0.52	**0.47**	7.7	6226	409
UVA16	17K	789K	**0.43**	341	**0.45**	**0.49**	2.4	130	387	**0.57**	400	**14**	19	310	8666	552
Virginia63	21K	698K	**0.29**	84	**0.29**	**0.29**	0.34	1.3	103	**0.26**	143	**1.1**	**1.1**	2.9	169	215
Wisconsin87	23K	835K	**0.17**	532	**0.18**	**0.18**	9.7	19	612	**0.31**	664	**47**	**43**	1323	292	924
wosn-friends	63K	817K	**0.16**	375	**0.30**	0.34	21	10.0	438	**1.0**	533	**54**	**55**	895	259	1260

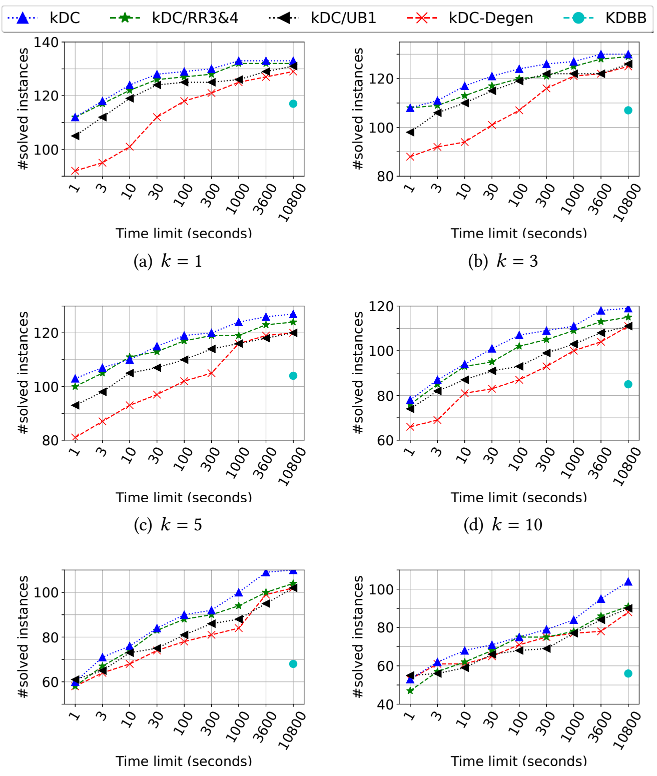
solves 5, 8, and 14 more instances than kDC/UB1&RR3&4, respectively, for $k = 3, 5$ and 10 on the Facebook graphs collection with a time limit of 100 seconds; this demonstrates that **RR3** and **RR4** are effective in these settings when **UB1** is not applied.

Secondly, we evaluate kDC against kDC/UB1 which is kDC without applying our upper bound **UB1** as introduced in Section 3.2.1. The results are also reported in Figures 7 and 8 and Table 3. We can see that kDC consistently outperforms kDC/UB1, and the improvement can be large, especially on the Facebook graphs. This demonstrates that our upper bound **UB1** is effective in improving the performance of kDC. Also, we would like to remark that **UB3** is the upper bound proposed in KDBB [16], and is also used in kDC. Thus, our upper bound **UB1** is also tighter than the one proposed in [16], as otherwise, the performance of kDC would be similar to that of kDC/UB1.

Thirdly, we compare kDC with kDC-Degen which is kDC invoking Degen to compute the initial solution at Line 1 of Algorithm 2 and without applying the reduction rule **RR6** at Line 2 of Algorithm 2. As a result, the preprocessing of kDC-Degen (i.e., Lines 1–2 of Algorithm 2) takes only $O(m)$ time, in contrast to the $O(\delta(G) \times m)$ preprocessing time of kDC. The experimental results are again shown in Figures 7 and 8 and Table 3. We can see that kDC consistently outperforms kDC-Degen, and the gap is huge when both k and the time limit are small, e.g., when $k \leq 5$ and the time limit is at most 10 seconds. To explain this, we also show the

Table 4: The difference of preprocessing results between kDC **and** kDC-Degen **(**C^0 **denotes the initial solution obtained by Line 1 of Algorithm 2;** n^0 **and** m^0 **respectively denote the number of vertices and the number of edges in the reduced graph obtained by Line 2 of Algorithm 2; subscripts denote the algorithms with** kDC-Degen **being abbreviated as** kDC-D**)**

	Real-world graphs			Facebook graphs		
	$\frac{\lvert C^0_{\text{kDC}}\rvert}{\lvert C^0_{\text{kDC-D}}\rvert}$	$\frac{n^0_{\text{kDC}}}{n^0_{\text{kDC-D}}}$	$\frac{m^0_{\text{kDC}}}{m^0_{\text{kDC-D}}}$	$\frac{\lvert C^0_{\text{kDC}}\rvert}{\lvert C^0_{\text{kDC-D}}\rvert}$	$\frac{n^0_{\text{kDC}}}{n^0_{\text{kDC-D}}}$	$\frac{m^0_{\text{kDC}}}{m^0_{\text{kDC-D}}}$
$k = 1$	1.19	0.27	0.26	1.30	0.03	0.02
$k = 3$	1.15	0.47	0.45	1.26	0.04	0.03
$k = 5$	1.13	0.52	0.52	1.24	0.06	0.04
$k = 10$	1.11	0.63	0.63	1.21	0.11	0.08
$k = 15$	1.09	0.68	0.69	1.19	0.16	0.13
$k = 20$	1.08	0.73	0.74	1.18	0.23	0.19

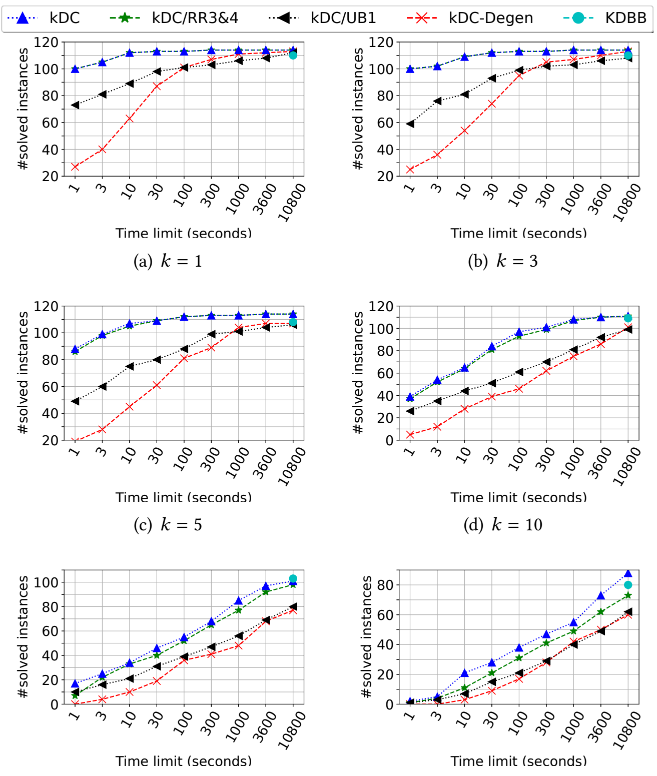


(a) $k = 1$ (b) $k = 3$

(c) $k = 5$ (d) $k = 10$

(e) $k = 15$ (f) $k = 20$

Figure 7: Number of solved instances for real-world graphs (vary time limit, best viewed in color)



(a) $k = 1$ (b) $k = 3$

(c) $k = 5$ (d) $k = 10$

(e) $k = 15$ (f) $k = 20$

Figure 8: Number of solved instances for Facebook graphs (vary time limit, best viewed in color)

Table 5: The (average and maximum) ratio of the maximum k-defective clique size over the maximum clique size for each of the three graph collections

	Real-world graphs		Facebook graphs		DIMACS10&SNAP	
	Avg Ratio	Max Ratio	Avg Ratio	Max Ratio	Avg Ratio	Max Ratio
$k = 1$	1.067	1.5	1.032	1.25	1.046	1.200
$k = 3$	1.144	2	1.083	1.5	1.107	1.400
$k = 5$	1.201	2	1.118	1.67	1.169	1.600
$k = 10$	1.314	2.5	1.170	1.75	1.243	1.800
$k = 15$	1.422	3	1.223	2	1.313	2.000
$k = 20$	1.516	3.5	1.264	2.25	1.370	2.200

difference of preprocessing results between kDC and kDC-Degen (i.e., Lines 1–2 of Algorithm 2) in Table 4. We can see that kDC computes a larger initial solution and a smaller reduced graph than kDC-Degen; the improvement is more significant when k is small.

In summary, each of these additional techniques (i.e., reduction rules **RR3** and **RR4**, upper bound **UB1**, and computing a large initial solution) improves the practical efficiency of kDC.

4.3 Properties of Maximum k-Defective Clique

In this subsection, we analyze the properties of maximum k-defective clique. Firstly, we compare the maximum k-defective clique size, computed by kDC, with the maximum clique size, computed by MC-BRB [7] [8], on the three graph collections. For each k and each of the three graph collections, the results on the average and maximum ratio of $\frac{\text{maximum } k\text{-defective clique size}}{\text{maximum clique size}}$ over all graphs that kDC finishes within 3 hours are reported in Table 5 (the total number of such graphs can be found in Table 2); we remark that MC-BRB successfully finds the maximum clique for all the graphs within the time limit. From Table 5, we can see that on the real-world graphs collection, the maximum k-defective clique size is on average 31% (and maximum 150%) larger than the maximum clique size for $k = 10$, and is on average 51% (and maximum 250%) larger for $k = 20$. This demonstrates that the relaxation of k-defective clique indeed enables us to find larger near-cliques.

Table 6: Number of graphs where the maximum k-defective clique is an extension of a maximum clique

	Real-world	Facebook	DIMACS10&SNAP
$k = 1$	133	114	37
$k = 3$	124	93	30
$k = 5$	114	77	28
$k = 10$	105	70	28
$k = 15$	98	62	23
$k = 20$	94	61	24

Secondly, we look into the actual maximum k-defective clique and check (1) whether it is an extension of a maximum clique and (2) what fraction of its vertices have missing neighbors. Note that, the maximum k-defective clique in a graph is not unique, and the results reported here are based on the maximum k-defective clique found by kDC and thus are only for the testings that finish within

[7] https://lijunchang.github.io/MC-BRB/

Table 7: Average percentage of vertices that are not fully connected in the maximum k-defective clique

	Real-world	Facebook	DIMACS10&SNAP
$k = 1$	19.2%	6.1%	16.9%
$k = 3$	33.7%	15.9%	32.3%
$k = 5$	43.3%	23.0%	46.6%
$k = 10$	52.5%	34.4%	56.8%
$k = 15$	59.5%	43.7%	64.7%
$k = 20$	62.9%	50.3%	65.9%

the time limit of 3 hours. The results are shown in Tables 6 and 7, respectively. From Table 6, we can see that for many, although not all, of the graphs, the maximum k-defective clique found by kDC is an extension of the maximum clique; specifically, the lowest fraction is $\frac{62}{101} \approx 61.4\%$ which is achieved for $k = 15$ on the Facebook graphs collection. Nevertheless, we have shown in Table 5 that maximum k-defective cliques are larger than maximum cliques. From Table 7, we can see that the percentage of vertices that are not fully connected in a maximum k-defective clique increases along with k, which is as expected. For $k \geq 10$ on the real-world graphs collection, more than half of the vertices in a maximum k-defective clique have missing neighbors in the k-defective clique.

5 RELATED WORK

The study of k-defective clique computation is still in its early stage. The first exact algorithm for computing the maximum k-defective clique was proposed in [44], which is based on the Russian doll search [45], a solver for general constraint optimization problems. The algorithm of [44] was then improved in [18] with new preprocessing rules as well as a better implementation. A branch-and-price framework was designed in [17]. A continuous cubic formulation was established in [39], which generalizes the Motzkin-Straus formulation from the maximum clique problem to the maximum k-defective clique problem; however, only heuristic algorithms are designed in [39]. Chen et al. [11] proposed the MADEC$^+$ algorithm whose time complexity beats the trivial $O^*(2^n)$ time complexity, and developed a graph coloring-based upper bound as well as other pruning techniques. The KDBB algorithm proposed in [16] is the currently fastest algorithm in practice, but its time complexity is the trivial $O^*(2^n)$. In this paper we proposed the kDC algorithm which not only has a better time complexity but also runs significantly faster in practice than all existing algorithms.

The problem of (approximately) counting all k-defective cliques of a particular size, for the special cases of $k = 1$ and 2, was recently formulated and studied in [21]. As the property of k-defective clique is *hereditary*, the number of k-defective cliques could explode drastically when the maximum k-defective clique size increases. Thus, the maximum k-defective clique size may provide a rough indication on the counting results. In addition, the pruning techniques proposed in this paper may speed up the enumeration and counting of *large* k-defective cliques.

Another related problem is maximum clique computation, which has been extensively studied both theoretically and practically. From a theoretical perspective, the worst case time complexity has been gradually improved from $O^*(1.4422^n)$ to $O^*(1.2599^n)$ [41], $O^*(1.2346^n)$ [22], and $O^*(1.2108^n)$ [32], with the state of the art being $O^*(1.1888^n)$ [33]; however, these algorithms are of theoretical interests only and have not been implemented. On the other hand, a plethora of practical algorithms, without caring about the time complexity analysis, have also been designed and implemented, e.g., [7, 8, 25, 26, 29, 30, 34, 36, 42, 43, 47]. For these algorithms, upper bounds have been demonstrated to be critical for the practical efficiency, and the most successful upper bounds are based on graph coloring and MaxSAT reasoning. However, these techniques cannot be easily extended to compute the maximum k-defective clique for $k \geq 1$, despite that k-defective clique is a relaxation of clique and 0-defective cliques are just cliques. For example, it was attempted in [11] to adapt the graph coloring to compute an upper bound of the maximum k-defective clique size, but as we demonstrated, the adaptation failed to compute a tight upper bound and is not effective in improving the efficiency. In contrast, we in this paper proposed a much tighter upper bound based on graph coloring.

6 FINDING TOP-r k-DEFECTIVE CLIQUES

In this section, we briefly discuss how to extend our techniques to two variants of finding top-r k-defective cliques. A thorough investigation of these problems is beyond the scope of this paper, and will be our future work.

Firstly, our techniques can be extended to find top-r maximal k-defective cliques, i.e., find the r maximal k-defective cliques that are largest. To do so, we will need to modify our algorithm to enumerate all large maximal k-defective cliques. Specifically, we will need to (1) change the $d_g(u)$ condition of **RR2** to $d_g(u) \geq |V(g)|-1$, (2) store in C the set of r currently found largest maximal k-defective cliques rather than just the single largest one, (3) change the lower bound lb used in **RR3**–**RR6** to be the size of the smallest k-defective clique in C. Due to the first change, the time complexity would be $O^*(\gamma_{2k}^n)$, the same as the maximum k-defective clique computation algorithm of [11].

Secondly, our techniques can be extended to find top-r diversified k-defective cliques, i.e., find r k-defective cliques that collectively cover/contain the most number of distinct vertices. Specifically, we iteratively conduct the following until r k-defective cliques have been reported or the graph becomes empty:

(1) find the maximum k-defective clique C in the current graph by invoking kDC,
(2) remove C from the current graph.

Note that, this approach may not find the optimal result, but the reported result provides a $(1 - \frac{1}{e})$-approximation guarantee. The time complexity is simply r times that of kDC.

7 CONCLUSION

In this paper, we advanced the state of the art for the problem of exact maximum k-defective clique computation, in terms of both worst case time complexity and practical performance. In specific, we first developed a general framework kDC based on our newly designed branching rule **BR** and reduction rules **RR1** and **RR2**. We proved that our framework beats the trivial time complexity of $O^*(2^n)$ and achieves a better time complexity than all existing algorithms. Then to make kDC practically efficient, we further proposed a new upper bound **UB1**, two reduction rules **RR3** and **RR4**, as well as an algorithm for efficiently computing a large initial solution. Extensive empirical studies on three benchmark graph collections with 290 graphs in total demonstrated the practical superiority of kDC over the existing algorithms.

ACKNOWLEDGMENTS

The author is supported by the Australian Research Council Fundings of FT180100256 and DP220103731.

REFERENCES

[1] James Abello, Mauricio G. C. Resende, and Sandra Sudarsky. 2002. Massive Quasi-Clique Detection. In *Proc. of LATIN'02 (Lecture Notes in Computer Science, Vol. 2286)*. Springer, 598–612.

[2] Mohiuddin Ahmed, Abdun Naser Mahmood, and Md Rafiqul Islam. 2016. A survey of anomaly detection techniques in financial domain. *Future Generation Computer Systems* 55 (2016), 278–288.

[3] Albert Angel, Nick Koudas, Nikos Sarkas, Divesh Srivastava, Michael Svendsen, and Srikanta Tirthapura. 2014. Dense subgraph maintenance under streaming edge weight updates for real-time story identification. *VLDB J.* 23, 2 (2014), 175–199.

[4] Balabhaskar Balasundaram, Sergiy Butenko, and Illya V. Hicks. 2011. Clique Relaxations in Social Network Analysis: The Maximum *k*-Plex Problem. *Operations Research* 59, 1 (2011), 133–142.

[5] Punam Bedi and Chhavi Sharma. 2016. Community detection in social networks. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 6, 3 (2016), 115–135.

[6] Jean-Marie Bourjolly, Gilbert Laporte, and Gilles Pesant. 2002. An exact algorithm for the maximum k-club problem in an undirected graph. *Eur. J. Oper. Res.* 138, 1 (2002), 21–28.

[7] Randy Carraghan and Panos M. Pardalos. 1990. An Exact Algorithm for the Maximum Clique Problem. *Oper. Res. Lett.* 9, 6 (Nov. 1990), 375–382.

[8] Lijun Chang. 2019. Efficient Maximum Clique Computation over Large Sparse Graphs. In *Proc. of KDD'19*. 529–538.

[9] Lijun Chang. 2020. Efficient maximum clique computation and enumeration over large sparse graphs. *VLDB J.* 29, 5 (2020), 999–1022.

[10] Lijun Chang and Lu Qin. 2018. *Cohesive Subgraph Computation over Large Sparse Graphs*. Springer Series in the Data Sciences.

[11] Xiaoyu Chen, Yi Zhou, Jin-Kao Hao, and Mingyu Xiao. 2021. Computing maximum k-defective cliques in massive graphs. *Comput. Oper. Res.* 127 (2021), 105131.

[12] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2001. *Introduction to Algorithms*. McGraw-Hill Higher Education.

[13] Qiangqiang Dai, Rong-Hua Li, Hongchao Qin, Meihao Liao, and Guoren Wang. 2022. Scaling Up Maximal *k*-plex Enumeration. In *Proc. of CIKM'22*. 345–354.

[14] 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).

[15] Fedor V. Fomin and Dieter Kratsch. 2010. *Exact Exponential Algorithms*. Springer.

[16] Jian Gao, Zhenghang Xu, Ruizhi Li, and Minghao Yin. 2022. An Exact Algorithm with New Upper Bounds for the Maximum k-Defective Clique Problem in Massive Sparse Graphs. In *Proc. of AAAI'22*. 10174–10183.

[17] Timo Gschwind, Stefan Irnich, Fabio Furini, and Roberto Wolfler Calvo. 2021. A Branch-and-Price Framework for Decomposing Graphs into Relaxed Cliques. *INFORMS J. Comput.* 33, 3 (2021), 1070–1090.

[18] Timo Gschwind, Stefan Irnich, and Isabel Podlinski. 2018. Maximum weight relaxed cliques and Russian Doll Search revisited. *Discret. Appl. Math.* 234 (2018), 131–138.

[19] Johan Håstad. 1996. Clique is Hard to Approximate Within $n^{1\text{-epsilon}}$. In *Proc. of FOCS'96*. 627–636.

[20] Shweta Jain and C. Seshadhri. 2020. The Power of Pivoting for Exact Clique Counting. In *Proc. WSDM'20*. ACM, 268–276.

[21] Shweta Jain and C. Seshadhri. 2020. Provably and Efficiently Approximating Near-cliques using the Turán Shadow: PEANUTS. In *Proc. of WWW'20*. ACM / IW3C2, 1966–1976.

[22] Tang Jian. 1986. An $O(2^{0.304n})$ Algorithm for Solving Maximum Independent Set Problem. *IEEE Trans. Computers* 35, 9 (1986), 847–851.

[23] Richard M. Karp. 1972. Reducibility Among Combinatorial Problems. In *Proc. of CCC'72*. 85–103.

[24] Victor E. Lee, Ning Ruan, Ruoming Jin, and Charu C. Aggarwal. 2010. A Survey of Algorithms for Dense Subgraph Discovery. In *Managing and Mining Graph Data*. Advances in Database Systems, Vol. 40. Springer, 303–336.

[25] 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*.

[26] 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.

[27] Ronghua Li, Sen Gao, Lu Qin, Guoren Wang, Weihua Yang, and Jeffrey Xu Yu. 2020. Ordering Heuristics for k-clique Listing. *Proc. VLDB Endow.* 13, 11 (2020), 2536–2548.

[28] David W. Matula and Leland L. Beck. 1983. Smallest-Last Ordering and clustering and Graph Coloring Algorithms. *J. ACM* 30, 3 (1983), 417–427.

[29] Panos M. Pardalos and Jue Xue. 1994. The maximum clique problem. *J. global Optimization* 4, 3 (1994), 301–328.

[30] 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.

[31] Jeffrey Pattillo, Nataly Youssef, and Sergiy Butenko. 2013. On clique relaxation models in network analysis. *Eur. J. Oper. Res.* 226, 1 (2013), 9–18.

[32] J. M. Robson. 1986. Algorithms for Maximum Independent Sets. *J. Algorithms* 7, 3 (1986), 425–440.

[33] J. M. Robson. 2001. Finding a maximum independent set in time $O(2^{n/4})$. https://www.labri.fr/perso/robson/mis/techrep.html.

[34] 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).

[35] H. Sachs. 1963. Regular Graphs with Given Girth and Restricted Circuits. *Journal of the London Mathematical Society* s1-38, 1 (1963), 423–429.

[36] 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.

[37] Stephen B. Seidman. 1983. Network structure and minimum degree. *Social Networks* 5, 3 (1983), 269 – 287.

[38] Hanif D. Sherali, J. Cole Smith, and Antonio A. Trani. 2002. An Airspace Planning Model for Selecting Flight-plans Under Workload, Safety, and Equity Considerations. *Transp. Sci.* 36, 4 (2002), 378–397.

[39] Vladimir Stozhkov, Austin Buchanan, Sergiy Butenko, and Vladimir Boginski. 2022. Continuous cubic formulations for cluster detection problems in networks. *Math. Program.* 196, 1 (2022), 279–307.

[40] Apichat Suratanee, Martin H Schaefer, Matthew J Betts, Zita Soons, Heiko Mannsperger, Nathalie Harder, Marcus Oswald, Markus Gipp, Ellen Ramminger, Guillermo Marcus, et al. 2014. Characterizing protein interactions employing a genome-wide siRNA cellular phenotyping screen. *PLoS computational biology* 10, 9 (2014), e1003814.

[41] Robert Endre Tarjan and Anthony E. Trojanowski. 1977. Finding a Maximum Independent Set. *SIAM J. Comput.* 6, 3 (1977), 537–546.

[42] Etsuji Tomita. 2017. Efficient Algorithms for Finding Maximum and Maximal Cliques and Their Applications. In *Proc. of WALCOM'17*. 3–15.

[43] 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.

[44] Svyatoslav Trukhanov, Chitra Balasubramaniam, Balabhaskar Balasundaram, and Sergiy Butenko. 2013. Algorithms for detecting optimal hereditary structures in graphs, with application to clique relaxations. *Comput. Optim. Appl.* 56, 1 (2013), 113–130.

[45] Gérard Verfaillie, Michel Lemaître, and Thomas Schiex. 1996. Russian Doll Search for Solving Constraint Optimization Problems. In *Proc. of AAAI'96*. AAAI Press / The MIT Press, 181–187.

[46] Jia Wang and James Cheng. 2012. Truss Decomposition in Massive Networks. *PVLDB* 5, 9 (2012).

[47] Jingen Xiang, Cong Guo, and Ashraf Aboulnaga. 2013. Scalable maximum clique computation using mapreduce. In *Proc. of ICDE'13*. 74–85.

[48] Mihalis Yannakakis. 1978. Node- and Edge-Deletion NP-Complete Problems. In *Proc. of STOC'78*. ACM, 253–264.

[49] Haiyuan Yu, Alberto Paccanaro, Valery Trifonov, and Mark Gerstein. 2006. Predicting interactions in protein networks by completing defective cliques. *Bioinform.* 22, 7 (2006), 823–829.

[50] Yi Zhou, Jingwei Xu, Zhenyu Guo, Mingyu Xiao, and Yan Jin. 2020. Enumerating Maximal *k*-Plexes with Worst-Case Time Guarantee. In *Proc. of AAAI'20*. 2442–2449.