Maximum Defective Clique Computation: Improved Time Complexities and Practical Performance

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

ABSTRACT

k-defective clique is a relaxation of the well-studied clique structure, by allowing up-to k edges missing from a clique. The problem of finding a k-defective clique with the largest number of vertices, although being NP-hard, has been receiving increasing interests recently, with advancements in both the theoretical time complexity and practical efficiency. The state-of-the-art time complexity is $O^*(\gamma_k^n)$, where O^* ignores polynomial factors, n is the number of vertices in the input graph G, and $\gamma_k < 2$ is a constant that only depends on k. In this paper, we first prove, through a more refined and non-trivial analysis, that the time complexity of an existing algorithm can actually be bounded by $O^*(\gamma_{k-1}^n)$, where $\gamma_{k-1} < \gamma_k$. Then, by utilizing the diameter-two property of large k-defective cliques, we show that for graphs with maximum k-defective clique sizes $\omega_k(G) \geq k+2$, a maximum k-defective clique can be found in $O^*((\alpha\Delta)^{k+2}\gamma_{k-1}^\alpha)$ time when using the degeneracy parameterization α and in $O^*((\alpha\Delta)^{k+2}(k+1)^{\alpha+k+1-\omega_k(G)})$ time when using the degeneracy-gap parameterization $\alpha+k+1-\omega_k(G)$; here, α and Δ are the degeneracy and maximum degree of G, respectively. Note that, most real graphs satisfy $\omega_k(G) \geq k+2$ and $\alpha \ll n$. Lastly, to improve the practical performance, we design a new degree-sequence-based reduction rule that can be efficiently applied, and theoretically demonstrate its effectiveness compared with the existing reduction rules. Extensive empirical studies on three benchmark graph collections, containing 290 graphs in total, show that our algorithm is also practically efficient, by outperforming all existing algorithms by several orders of magnitude. We remark that our proving techniques for reducing the base from γ_k to γ_{k-1} and our general principle of designing a new reduction rule may also be beneficial to other problems.

PVLDB Reference Format:
Lijun Chang. Maximum Defective Clique Computation: Improved Time Complexities and Practical Performance. PVLDB, 18(2): 200 - 212, 2024.
doi:10.14778/3705829.3705839

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://lijunchang.github.io/kDC-two/.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 18, No. 2 ISSN 2150-8097.
doi:10.14778/3705829.3705839

1 INTRODUCTION

Graphs have been widely used to capture the relationship between entities in applications such as social media, communication network, e-commerce, and cybersecurity. Identifying dense subgraphs from those real-world graphs, which are usually globally sparse (e.g., have a small average degree), is a fundamental problem and has received a lot of attention [10, 25]. Dense subgraphs may correspond to communities in social networks [4], protein complexes in biological networks [41], and anomalies in financial networks [2]. The clique model, requiring every pair of vertices to be directly connected by an edge, represents the densest structure that a subgraph can be. As a result, clique related problems have been extensively studied, e.g., theoretical aspect of maximum clique computation [23, 34, 35, 42], practical aspect of maximum clique computation [6, 7, 26, 27, 31, 32, 36, 38, 43, 44, 47], maximal clique enumeration [11, 13, 15], and k-clique counting and enumeration [9, 21, 28].

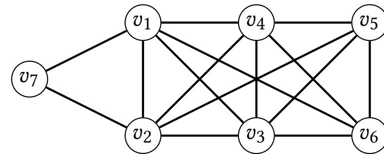

Figure 1: Defective clique

Requiring every pair of vertices to be explicitly connected by an edge however is often too restrictive in practice, by noticing that data may be noisy or incomplete and/or the data collection process may introduce errors [33]. In view of this, various clique relaxation models have been formulated and studied in the literature, such as quasi-clique [1], plex [3], club [5], and defective clique [49]. A subgraph with c vertices is a k-defective clique if it has at least $\binom{c}{2} - k$ edges, i.e., it misses at most k edges from being a clique. A k-defective clique is usually referred to by its vertices, since maximal k-defective cliques are vertex-induced subgraphs. Consider the graph in Figure 1, $\{v_1, \ldots, v_4\}$ is a maximum clique, $\{v_1, \ldots, v_4, v_7\}$ is a *maximal* 2-defective clique, and $\{v_1, \ldots, v_6\}$ is a *maximum* 2-defective clique that maximizes the number of vertices. Finding large defective cliques has applications in biological networks [49], cluster detection [14, 40], transportation science [39], and social network analysis [19, 22]. For example, missing links in a large defective clique are good candidates for predicting implicit interactions between proteins in biological networks in [49]. Also, defective clique has been used for predicting potential collaborations between authors based on the DBLP data and for finding closely related financial instruments on stock markets [14].

In this paper, we study the problem of finding a k-defective clique with the largest number of vertices, which is NP-hard [48].

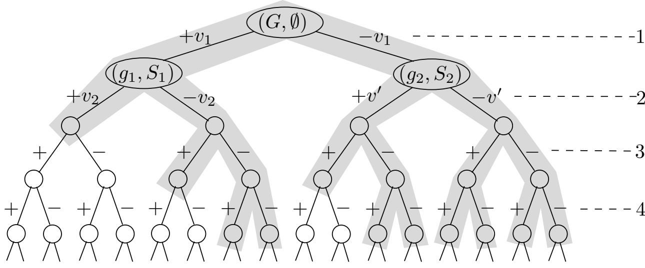

Figure 2: Search tree $\mathcal{T}$ of a backtracking algorithm

The state-of-the-art time complexity, beating the trivial $O^*(2^n)$, is achieved by the kDC algorithm [8], running in $O^*(\gamma_k^n)$ time where $\gamma_k < 2$ is the largest real root of the equation $x^{k+3} - 2x^{k+2} + 1 = 0$ and n is the number of vertices in the input graph G. kDC is a branch-and-bound (aka. backtracking) algorithm. Let $\mathcal{T}$ be the backtracking search tree (see Figure 2) where each node represents a backtracking instance (g, S) with g being a subgraph of G and $S \subseteq V(g)$ a k-defective clique (i.e., S is a partial solution). The left child of (g, S) adds the *branching vertex* v to S (i.e., $+v$), while the right child removes v from g (i.e., $-v$). It suffices to bound the number of leaf nodes of $\mathcal{T}$, as the O^* notation ignores polynomial factors. kDC achieves its time complexity by ① deterministically processing vertices that have up-to one non-neighbor (by reduction rule **RR2** of [8]), and ② greedily ordering vertices (by branching rule **BR** of [8]) such that *a length-$(k+2)$ prefix of the ordering induces more than k missing edges*. ② ensures the time complexity since we only need to consider up-to $k+2$ prefixes when enumerating the prefixes that can be added to the solution, while ① makes ② possible.

We first aim to reduce the *base* (i.e., γ_k) of the exponential time complexity. This is challenging, see Section 4.1 for details. Nevertheless, we manage to prove that the time complexity of kDC [8] can be bounded by $O^*(\gamma_{k-1}^n)$, by using different analysis techniques for different backtracking instances. Our general idea is that if at least one branching vertex (i.e., previously selected by **BR**) has been added to S (e.g., (g_1, S_1) in Figure 2), then **BR** computes an ordering of $V(g) \setminus S$ such that the union of S and *a length-$(k+1)$ prefix of the ordering induces more than k missing edges* (Lemma 4.1); thus, the number of leaf nodes of $\mathcal{T}$ rooted at (g, S) can be shown *by induction* to be at most $\gamma_{k-1}^{|V(g)\setminus S|}$ (Lemma 4.2). Otherwise (e.g., (g_2, S_2)), we prove *non-inductively* that the number of leaf nodes is at most $2 \cdot \gamma_{k-1}^{|V(g)\setminus S|}$ by introducing the coefficient 2 (Lemma 4.3).

Secondly, we show that the *exponent* of the time complexity can be reduced for graphs whose maximum k-defective cliques $\omega_k(G)$ are large. It is known that any k-defective clique of size $\geq k+2$ has a diameter at most two. Thus, for a backtracking instance (g, S), once a vertex u is added to S, we can remove from g all vertices whose shortest distances (computed in g) to u are larger than two. Let $(v_1, v_2, \ldots, v_n)$ be a degeneracy ordering of $V(G)$. We process each vertex v_i by assuming that it is the first vertex of the degeneracy ordering that is in a maximum k-defective clique; note that, at least one of these n assumptions will be true, and thus we can find a maximum k-defective clique. By using the same proving techniques developed above, it can be shown that the search tree of processing v_i has at most $2 \cdot \gamma_{k-1}^{\alpha\Delta}$ leaf nodes since we only need to consider v_i's neighbors and two-hop neighbors that come later than v_i in the degeneracy ordering; here α and Δ are the degeneracy and maximum degree of G, respectively. Through a more refined analysis, we show that the number of leaf nodes is also bounded by $O((\alpha\Delta)^k \gamma_{k-1}^{\alpha})$. Consequently, a maximum k-defective clique of G can be found in $O(n \times (\alpha\Delta)^{k+2} \times \gamma_{k-1}^{\alpha})$ time when $\omega_k(G) \geq k+2$. Note that, most real graphs satisfy $\omega_k(G) \geq k+2$ and $\alpha \ll n$. Furthermore, we show that a maximum k-defective clique can also be found in $O^*((\alpha\Delta)^{k+2} \times (k+1)^{\alpha+k+1-\omega_k(G)})$ time when using the degeneracy-gap parameterization $\alpha + k + 1 - \omega_k(G)$ which is at most $\alpha - 1$ since $\omega_k(G) \geq k+2$.

Thirdly, to improve the practical performance, we design a new reduction rule **RR3** based on the degree-sequence-based upper bound **UB** proposed in [18]. However, instead of using **UB** to prune instances after generating them as done in the existing works [8, 18], we remove vertex $u \in V(g) \setminus S$ from g if an upper bound of $(g, S \cup u)$ is no larger than the current best solution size lb. Note that, rather than computing the exact upper bound for $(g, S \cup u)$, we test whether the upper bound is larger than lb or not. The latter can be conducted more efficiently and without generating $(g, S \cup u)$; moreover, computation can be shared between the testing for different vertices of $V(g) \setminus S$. We show that with linear time preprocessing, the upper bound testing for all vertices $u \in V(g) \setminus S$ can be conducted in totally linear time. In addition, we theoretically demonstrate that **RR3** is more effective than the existing reduction rules, e.g., the degree-sequence-based reduction rule and second-order reduction rule proposed in [8].

Contributions. Our main contributions are as follows.

- We improve the time complexity of the state-of-the-art algorithm kDC [8] from $O^*(\gamma_k^n)$ to $O^*(\gamma_{k-1}^n)$, through a more refined and non-trivial analysis. (Section 4).
- We show that for graphs with $\omega_k(G) \geq k+2$, a maximum k-defective clique can be found in $O^*((\alpha\Delta)^{k+2} \times \gamma_{k-1}^{\alpha})$ time and also in $O^*((\alpha\Delta)^{k+2} \times (k+1)^{\alpha+k+1-\omega_k(G)})$ time by using different parameterizations. (Section 5)
- We design a new degree-sequence-based reduction rule **RR3** that can be conducted in linear time, and theoretically demonstrate its effectiveness compared with the existing reduction rules. (Section 6)

We conduct extensive empirical studies on three benchmark collections with 290 graphs in total to evaluate our techniques (Section 7). The results show that (1) our algorithm solves 34 and 36 more graph instances than the most recent algorithms kDC [8] and KD-Club [24], respectively, for a time limit of 3 hours and $k = 15$; (2) on the 33 real-world graphs that have more than 100,000 vertices, our algorithm is on average (with geometric mean) one order of magnitude faster than kDC and two orders of magnitude faster than KD-Club, for $k = 15$.

2 PROBLEM DEFINITION

We consider a large *unweighted*, *undirected* and *simple* graph $G = (V, E)$ and refer to it simply as a graph; here, V is the vertex set and E is the edge set. The numbers of vertices and edges of G are denoted by $n = |V|$ and $m = |E|$, respectively. An undirected edge between u and v is denoted by (u, v) and (v, u). The set of edges that are missing from G is called the set of ***non-edges*** (or missing edges)

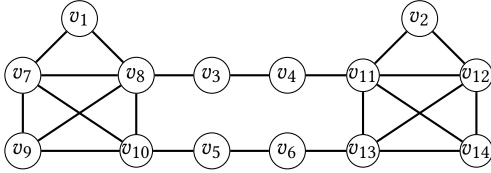

Figure 3: An example graph

of G and denoted by $\overline{E}$, i.e., $(u,v) \in \overline{E}$ if $u \neq v$ and $(u,v) \notin E$. The set of u's neighbors in G is denoted $N_G(u) = \{v \in V \mid (u,v) \in E\}$, and the *degree* of u in G is $d_G(u) = |N_G(u)|$; similarly, the set of u's ***non-neighbors*** in G is denoted $\overline{N}_G(u) = \{v \in V \mid (u,v) \in \overline{E}\}$. Note that ***a vertex is neither a neighbor nor a non-neighbor of itself***. Given a vertex subset $S \subseteq V$, the set of edges *induced* by S is $E_G(S) = \{(u,v) \in E \mid u,v \in S\}$, the set of non-edges induced by S is $\overline{E}_G(S) = \{(u,v) \in \overline{E} \mid u,v \in S\}$, and the subgraph of G induced by S is $G[S] = (S, E_G(S))$. We denote the union of a set S and a vertex u by $S \cup u$, and the subtraction of u from S by $S \setminus u$. For presentation simplicity, we omit the subscript G from the notations when the context is clear, and abbreviate $N_{G[S \cup u]}(u) = N(u) \cap S$ as $N_S(u)$ and $\overline{N}_{G[S \cup u]}(u) = \overline{N}(u) \cap S$ as $\overline{N}_S(u)$. For an arbitrary graph g, we denote its sets of vertices, edges and non-edges by $V(g)$, $E(g)$ and $\overline{E}(g)$, respectively.

Definition 2.1 (k-Defective Clique). A graph g is a k-defective clique if it misses at most k edges from being a clique, i.e., $|E(g)| \geq \frac{|V(g)|(|V(g)|-1)}{2} - k$ or equivalently, $|\overline{E}(g)| \leq k$.

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, *we refer to a k-defective clique simply by its set of vertices*, and measure the size of a k-defective clique $S \subseteq V$ by its number of vertices, i.e., $|S|$. 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 S of G is a *maximal k-defective clique* if every proper superset of S 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; denote the size of the maximum k-defective clique of G by $\omega_k(G)$. Consider the graph in Figure 3, both $\{v_1, v_7, \ldots, v_8\}$ and $\{v_2, v_{11}, \ldots, v_{14}\}$ are maximum 2-defective cliques with $\omega_2(G) = 5$; that is, the maximum k-defective clique is not unique.

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

Note that, although we only aim to find one largest k-defective clique in this paper, our algorithm can be easily extended to find multiple large k-defective cliques. For example, we can iteratively find a largest k-defective clique and remove it from the graph; it can be shown that the first h reported results form an approximate solution to the diversified top-h defective clique problem with an approximation ratio of $(1 - \frac{1}{e})$ [8].

Frequently used notations are summarized in Table 1.

3 STATE OF THE ART

The problem of maximum k-defective clique computation is NP-hard [48]. The existing exact algorithms compute a maximum k-defective clique via *branch-and-bound* search (aka. *backtracking*).

Table 1: Frequently used notations

Notation	Meaning
$G = (V, E)$	a graph with vertex set V and edge set E
$\omega_k(G)$	the size of the maximum k-defective clique of G
$g = (V(g), E(g))$	a subgraph of G
$S \subseteq V$	a k-defective clique
(g, S)	a backtracking instance with $S \subseteq V(g)$
$N_S(u)$	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 *edges* induced by S
$\overline{E}(S)$	the set of *non-edges* induced by S
$\mathcal{T}, \mathcal{T}'$	search tree of backtracking algorithms
$I, I', I_0, I_1, \ldots$	nodes of the search tree $\mathcal{T}$ or $\mathcal{T}'$
$\|I\|$	the size of I, i.e., $\|V(I.g) \setminus I.S\|$
$\ell_{\mathcal{T}}(I)$	number of leaf nodes in the subtree of $\mathcal{T}$ rooted at I

Algorithm 1: Branch&Bound(g, S)

1. $(g', S') \leftarrow$ apply reduction rules **RR1** and **RR2** to (g, S);
2. **if** g' *is a k-defective clique* **then** update C^* by $V(g')$ and **return**;
3. $b \leftarrow$ choose a branching vertex from $V(g') \setminus S'$ based on **BR**;
4. Branch&Bound$(g', S' \cup b)$; /* Left branch includes b */;
5. Branch&Bound$(g' \setminus b, S')$; /* Right branch excludes b */;

Let (g, S) denote a backtracking instance, where g is a (sub-)graph (of the input graph G) and $S \subseteq V(g)$ is a k-defective clique in g. The goal of solving the instance (g, S) is to find a largest k-defective clique in the instance (i.e., in g and containing S); thus, solving the instance $(G, \emptyset)$ finds a maximum k-defective clique in G. To solve an instance (g, S), a backtracking algorithm selects a branching vertex $b \in V(g) \setminus S$, and then recursively solves two newly generated instances: one includes b into S, and the other removes b from g. For the base case that $S = V(g)$, S is the maximum k-defective clique in the instance. For example, Figure 2 shows a snippet of the backtracking search tree $\mathcal{T}$, where each node corresponds to a backtracking instance (g, S). The two newly generated instances are represented as the two children of the node, and the branching vertex is illustrated on the edge; for the sake of simplicity, Figure 2 only shows the branching vertices for the first two levels.

The state-of-the-art time complexity is achieved by kDC [8] which proposes a new branching rule and two reduction rules to achieve the time complexity. Specifically, kDC proposes the non-fully-adjacent-first branching rule **BR** preferring branching on a vertex that is not fully adjacent to S, and the excess-removal reduction rule **RR1** and the high-degree reduction rule **RR2**.

BR [8]. Given an instance (g, S), the branching vertex is selected as the vertex of $V(g) \setminus S$ that has at least one non-neighbor in S; if no such vertices exist, an arbitrary vertex of $V(g) \setminus S$ is chosen as the branching vertex.

RR1 [8]. Given an instance (g, S), if a vertex $u \in V(g) \setminus S$ satisfies $|\overline{E}(S \cup u)| > k$, we can remove u from g.

RR2 [8]. Given an instance (g, S), if a vertex $u \in V(g) \setminus S$ satisfies $|\overline{E}(S \cup u)| \leq k$ and $d_g(u) \geq |V(g)| - 2$, we can greedily add u to S.

The pseudocode of Branch&Bound is shown in Algorithm 1. Given an input (g, S), it first applies reduction rules **RR1** and **RR2** to reduce the instance (g, S) to a potentially smaller instance (g', S')

such that $V(g') \setminus S' \subseteq V(g) \setminus S$ (Line 1). If g' itself is a k-defective clique, then it updates C^* by $V(g')$ and backtrack (Line 2). Otherwise, it picks a branching vertex b based on the branching rule **BR** (Line 3), and generates two new Branch&Bound instances and go into recursion (Lines 4–5). kDC invokes Branch&Bound$(G, \emptyset)$ to find the maximum k-defective clique in G.

Time Complexity Analysis of kDC**.** The general idea of time complexity analysis is as follows. As polynomial factors are usually ignored in the time complexity analysis of exponential time algorithms, it is sufficient to bound the number of leaf nodes of the search tree (in Figure 2) inductively in a bottom-up fashion [17]. One way of bounding the number of leaf nodes of the subtree rooted at the node corresponding to instance (g, S) is to order $V(g) \setminus S$ in such a way that the longest prefix of the ordering that can be added to S without violating the k-defective clique definition is short and bounded. Specifically, let $(v_1, \ldots, v_l, v_{l+1}, \ldots)$ be an ordering of $V(g) \setminus S$ such that the longest prefix that can be added to S without violating the k-defective clique definition is $(v_1, \ldots, v_l)$; that is, $\{v_1, \ldots, v_l, v_{l+1}\} \cup S$ induces more than k non-edges. Then, we only need to generate $l + 1$ new instances/branches, corresponding to the first $l + 1$ prefixes, as shown in Figure 4: for the i-th (starting from 0) branch, we include $(v_1, \ldots, v_i)$ to S and remove v_{i+1} from g. Denote the i-th branch by (g_i, S_i). It holds that

- $|V(g_i) \setminus S_i| \leq |V(g) \setminus S| - (i + 1), \forall 0 \leq i \leq l.$

It can be shown by the techniques of [17] that the number of leaf nodes of the search tree is at most γ^n where $\gamma < 2$ is the largest real root of the equation $x^{l_{\max}+2} - 2x^{l_{\max}+1} + 1 = 0$ and $l_{\max}$ is the largest l among all non-leaf nodes. Thus, the smaller the value of $l_{\max}$, the better the time complexity.

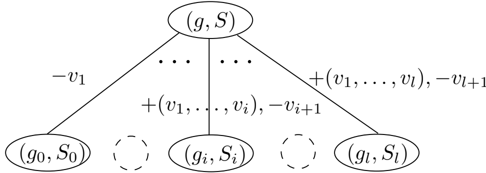

Figure 4: General idea of time complexity analysis

For kDC, $l_{\max} = k + 1$ and thus its time complexity is $O^*(\gamma_k^n)$ where $\gamma_k < 2$ is the largest real root of the equation $x^{k+3} - 2x^{k+2} + 1 = 0$ [8]; here, the O^* notation hides polynomial factors. Specifically, kDC orders $V(g) \setminus S$ by iteratively applying **BR**. That is, each time it appends to the ordering a vertex that has at least one non-neighbor in either S or the vertices already in the ordering; if no such vertices exist, an arbitrary vertex of $V(g) \setminus S$ is appended. It is proved in [8] that after exhaustively applying reduction rules **RR1** and **RR2**, the resulting instance (g, S) satisfies the condition:

- $|\overline{E}(S \cup u)| \leq k$ and $d_g(u) < |V(g)| - 2, \forall u \in V(g) \setminus S.$

i.e., **every vertex of** $V(g) \setminus S$ **has at least two non-neighbors in** g. Then, the worst-case scenario (for time complexity) is that the non-edges of $g[V(g) \setminus S]$ form a set of vertex-disjoint cycles; a length-$(k + 1)$ prefix of the ordering induces exactly k non-edges, and a length-$(k + 2)$ prefix induces more than k non-edges.

One may notice that Figure 2 illustrates a *binary* search tree while Figure 4 shows a *multi-way* search tree. Nevertheless, the above techniques can be used to analyze Figure 2 since a binary search tree can be (virtually) converted into an equivalent multi-way search tree, which is the way the time complexity of kDC was analyzed in [8]. That is, we could collapse a length-l path in Figure 2 to make it have $l + 1$ children. This will be more clear when we conduct our time complexity analysis in Lemma 4.2.

4 AN IMPROVED TIME COMPLEXITY

In this section, we improve the time complexity of maximum k-defective clique computation from $O^*(\gamma_k^n)$ to $O^*(\gamma_{k-1}^n)$. Before that, we first discuss the challenges of improving the time complexity.

4.1 Challenges

As discussed in Section 3, the smaller the value of $l_{\max}$, the better the time complexity. kDC [8] proves that $l_{\max} \leq k + 1$ by making all vertices of $V(g) \setminus S$ have at least two non-neighbors in g, which is achieved by **RR2**. In contrast, if all vertices of $V(g) \setminus S$ have exactly one non-neighbor in g, then $l_{\max}$ becomes $2k + 1$ and the time complexity is $O^*(\gamma_{2k}^n)$, which is the case of MADEC$^+$ [12]. It is natural to wonder whether the value of $l_{\max}$ can be reduced if we have techniques to make each vertex of $V(g) \setminus S$ have more (than two) non-neighbors in g. Specifically, let's consider the complement graph $\overline{g}$ of g: each edge of $\overline{g}$ corresponds to a non-edge of g. The question is whether the **BR** of kDC can guarantee $l_{\max} < k+1$ when $\overline{g}[V(g) \setminus S]$ has a minimum degree larger than two. Unfortunately, the answer is negative. It is shown in [37] that for any $r \geq 2$ and $s \geq 3$, there exists a graph in which each vertex has exactly r neighbors and the shortest cycle has length exactly s; these graphs are called (r, s)-graphs. Thus, when $\overline{g}[V(g) \setminus S]$ is an (r, s)-graph for $s \geq k + 2$, iteratively applying the branching rule **BR** may first identify vertices of the shortest cycle and it then needs a prefix of length $k + 2$ to cover $k + 1$ edges of $\overline{g}[V(g) \setminus S]$ (corresponding to $k + 1$ non-edges of g).

Alternatively, one may tempt to design a different branching rule than **BR** for finding a subset of $k + 1$ or fewer vertices $C \subseteq V(g) \setminus S$ such that $\overline{g}[C]$ has at least $k + 1$ edges. This however most likely cannot be conducted efficiently, by noting that it is NP-hard to find a densest k-subgraph (i.e., a subgraph with exactly k vertices and the most number of edges) when k is a part of the input [16].

4.2 Improving the Time Complexity

Despite the challenges and negative results mentioned in Section 4.1, we in this subsection show that the time complexity of maximum k-defective clique computation can be improved by conducting a more refined analysis of the existing algorithm kDC, without proposing any new algorithmic techniques.

We use the same terminologies and notations as [8] and consider the search tree $\mathcal{T}$ of (recursively) invoking Branch&Bound, as shown in Figure 2. To avoid confusion, nodes of the search tree are referred to by *nodes*, and vertices of a graph by *vertices*. Nodes of $\mathcal{T}$ are denoted by $I, I', I_0, I_1, \ldots$, and the graph g and the partial solution S of the Branch&Bound instance to which I corresponds are, respectively, denoted by $I.g$ and $I.S$. Note that *$I.g$ and $I.S$ denote the ones obtained after applying the reduction rules* at Lines 1–2 of Algorithm 1, where Line 2 is regarded as applying the reduction rule that if g' is a k-defective clique, then all vertices of $V(g') \setminus S'$

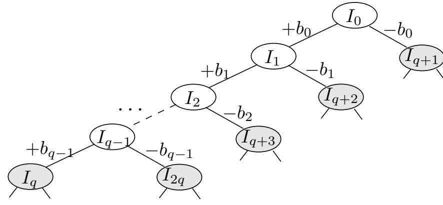

Figure 5: A left-deep walk in the search tree $\mathcal{T}$ starting from I_0; I_q is the first node such that $q \geq 1$ and $|V(g_q)| < |V(g_{q-1})|$

are added to S'. The size of I is measured by the number of vertices of $I.g$ that are not in $I.S$, i.e., $|I| = |V(I.g)| - |I.S| = |V(I.g) \setminus I.S|$. It is worth mentioning 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 $V(I'.g) \setminus I'.S$.
- $|I| = 0$ whenever I is a leaf node

For a non-leaf node I_0, we consider the path $(I_0, I_1, \ldots, I_q)$ that starts from I_0, always visits the left child in $\mathcal{T}$, and stops at I_q once $q \geq 1$ and $|V(g_q)| < |V(g_{q-1})|$; see Figure 5 for an example. Note that, the path is well defined since I_0 is a non-leaf node and thus $I_0.g$ is not a k-defective clique. For presentation simplicity, we abbreviate $I_i.g$ and $I_i.S$ as g_i and S_i, respectively, and denote the branching vertex selected for I_i by b_i. Let u be an arbitrary vertex of $V(g_{q-1}) \setminus V(g_q)$. Then, $(b_0, \ldots, b_{q-1}, u, \ldots)$ is an ordering of $V(g_0) \setminus S_0$ such that adding $(b_0, \ldots, b_{q-1}, u)$ to S_0 violates the k-defective clique definition. In the lemma below, we prove that $q \leq k$ if I_0 has at least one branching vertex being added.

Lemma 4.1. *If the non-leaf node I_0 has at least one branching vertex being added, then $q \leq k$.*

Proof. Let I_x be the last node, on the path $(I_0, I_1, \ldots, I_x, \ldots, I_{q-1})$, satisfying the condition that all vertices of $V(g_x) \setminus S_x$ are adjacent to all vertices of S_x, i.e., the branching vertex b_x selected for I_x has no non-neighbors in S_x. If such an I_x does not exist, then we have $|\overline{E}(S_{i+1})| \geq |\overline{E}(S_i)| + 1$ for all $0 \leq i < q$ (because the branching vertex added to S_{i+1} must bring at least one non-edge to $\overline{E}(S_{i+1})$), and consequently $q \leq |\overline{E}(S_q)| \leq k$. In the following, we assume that such an I_x exists, and prove that $|\overline{E}(S_x)| > x$ by considering two cases. Note that, each of the branching vertices $b_i \in \{b_0, \ldots, b_{x-1}\}$ that are added to S_x must have at least two non-neighbors in g_i (because of **RR2**) and all these non-neighbors are in S_x (according to the definitions of I_x and I_q).

- **Case-I**: $|\overline{E}(S_0)| \neq 0$. Then, the number of unique non-edges associated with $\{b_0, \ldots, b_{x-1}\}$ is at least x, and $|\overline{E}(S_x)| \geq |\overline{E}(S_0)| + x > x$.
- **Case-II**: $|\overline{E}(S_0)| = 0$. Let b_{-1} be the branching vertex added to S_0 from its parent (which exists according to the lemma statement). Then, b_{-1} has at least two non-neighbors in g_0; note that these non-neighbors will not be removed from g_0 by **RR1** since $|\overline{E}(S_0)| = 0$ and all vertices of $V(g_0) \setminus S_0$ are fully adjacent to vertices of $S_0 \setminus b_{-1}$. Thus, the number of non-edges between $\{b_{-1}, b_0, \ldots, b_{x-1}\}$ is at least $x + 1$, and hence $|\overline{E}(S_x)| > x$.

Then, according to the definition of I_x and our branching rule, for each i with $x + 1 \leq i < q$, the branching vertex b_i selected for I_i has at least one non-neighbor in S_i, and consequently,

$$|\overline{E}(S_q)| \geq |\overline{E}(S_x)| + (q - x - 1) > q - 1$$

Thus, the lemma follows from the fact that $|\overline{E}(S_q)| \leq k$. □

Let $\ell_{\mathcal{T}}(I)$ denote the number of leaf nodes in the subtree of $\mathcal{T}$ rooted at I. We prove in the lemma below that $\ell_{\mathcal{T}}(I) \leq \beta_k^{|I|}$ when at least one branching vertex has been added to I. Note that $\beta_k = \gamma_{k-1}$.

Lemma 4.2. *For any node I of $\mathcal{T}$ that has at least one branching vertex being added, it holds that $\ell_{\mathcal{T}}(I) \leq \beta_k^{|I|}$ where $1 < \beta_k < 2$ is the largest real root of the equation $x^{k+2} - 2x^{k+1} + 1 = 0$.*

Proof. We prove the lemma by induction. For the base case that I is a leaf node, it is trivial that $\ell_{\mathcal{T}}(I) = 1 \leq \beta_k^{|I|}$ since $\beta_k > 1$ and $|I| = 0$. For a non-leaf node I_0, let's consider the path $(I_0, I_1, \ldots, I_{q'})$ that starts from I_0, always visits the left child in the search tree $\mathcal{T}$, and stops at $I_{q'}$ once $q' \geq 1$ and $|I_{q'}| \leq |I_{q'-1}| - 2$. Note that $(I_0, \ldots, I_{q'})$ is a prefix of $(I_0, \ldots, I_q)$ since I_q satisfies the condition that $q \geq 1$ and $|I_q| \leq |I_{q-1}| - 2$. It is trivial that

$$\ell_{\mathcal{T}}(I_0) = \ell_{\mathcal{T}}(I_{q'}) + \ell_{\mathcal{T}}(I_{q'+1}) + \ell_{\mathcal{T}}(I_{q'+2}) + \cdots + \ell_{\mathcal{T}}(I_{2q'})$$

where $I_{q'+1}, I_{q'+2}, \ldots, I_{2q'}$ are the right child of $I_0, I_1, \ldots, I_{q'-1}$, respectively, as illustrated in Figure 5 (by replacing q with q'); *this is equivalent to converting a binary search tree to a multi-way search tree by collapsing the path $(I_0, \ldots, I_{q'-1})$ into a super-node that has $I_{q'}, I_{q'+1}, \ldots, I_{2q'}$ as its children.* To bound $\ell_{\mathcal{T}}(I_0)$, we need to bound q' and $|I_i|$ for $q' \leq i \leq 2q'$. Following from Lemma 4.1, we have

Fact 1. $q' \leq q \leq k$.

Also, according to the definition of the path, it holds that

$$\forall i \in [1, q'-1], \quad S_i = S_{i-1} \cup b_{i-1}, \quad V(g_i) = V(g_{i-1}) \qquad (1)$$

That is, the reduction rules at Line 1 of Algorithm 1 have no effect on I_i for $1 \leq i < q'$. Then, the following two facts hold.

Fact 2. $\forall i \in [q'+1, 2q'], |I_i| \leq |I_{i-q'-1}| - 1 \leq |I_0| + q' - i$.

Fact 3. $|I_{q'}| \leq |I_{q'-1}| - 2 \leq |I_0| - q' - 1$.

Based on Facts 1, 2 and 3, we have

$$\begin{aligned}
\ell_{\mathcal{T}}(I_0) &= \ell_{\mathcal{T}}(I_{q'+1}) + \ell_{\mathcal{T}}(I_{q'+2}) + \cdots + \ell_{\mathcal{T}}(I_{2q'}) + \ell_{\mathcal{T}}(I_{q'}) \\
&\leq \beta_k^{|I_{q'+1}|} + \beta_k^{|I_{q'+2}|} + \cdots + \beta_k^{|I_{2q'}|} + \beta_k^{|I_{q'}|} \\
&\leq \beta_k^{|I_0|-1} + \beta_k^{|I_0|-2} + \cdots + \beta_k^{|I_0|-q'} + \beta_k^{|I_0|-q'-1} \\
&\leq \beta_k^{|I_0|-1} + \beta_k^{|I_0|-2} + \cdots + \beta_k^{|I_0|-k} + \beta_k^{|I_0|-k-1}
\end{aligned}$$

where $\beta_k^{|I_0|-1} + \beta_k^{|I_0|-2} + \cdots + \beta_k^{|I_0|-k} + \beta_k^{|I_0|-k-1} \leq \beta_k^{|I_0|}$ if β_k is no smaller than the largest real root of the equation $x^{k+1} - x^k - \cdots - x - 1 = 0$ which is equivalent to the equation $x^{k+2} - 2x^{k+1} + 1 = 0$ [17]. The first few solutions to the equation are $\beta_1 = 1.619$, $\beta_2 = 1.840$, $\beta_3 = 1.928$, $\beta_4 = 1.966$, and $\beta_5 = 1.984$. □

In Lemma 4.2, we cannot bound $\ell_{\mathcal{T}}(I)$ by $\beta_k^{|I|}$ if no branching vertices have been added to I. Nevertheless, we prove in the lemma below that $\ell_{\mathcal{T}}(I) < 2 \cdot \beta_k^{|I|}$ holds for every node I of $\mathcal{T}$, by using a non-inductive proving technique.

Lemma 4.3. *For any node I of $\mathcal{T}$, it holds that $\ell_{\mathcal{T}}(I) < 2 \cdot \beta_k^{|I|}$.*

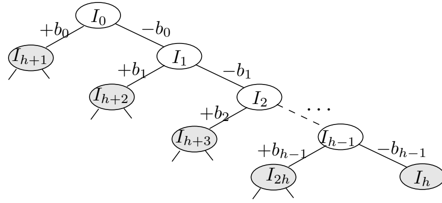

Figure 6: A right-deep walk in the search tree $\mathcal{T}$ starting from I_0 and stopping at a leaf node I_h

Proof. If I is a leaf node, the lemma is trivial. For a non-leaf node I_0, let's consider the path $(I_0, I_1, \ldots, I_h)$ that starts from I_0, always visits the right child in the search tree $\mathcal{T}$, and stops at a leaf node I_h; see Figure 6 for an example. Then, it holds that

$$\forall i \in [h+1, 2h], |I_i| \leq |I_{i-h-1}| - 1 \leq |I_0| + h - i$$

and $|I_h| = 0$. Moreover, for each $i \in [h+1, 2h]$, I_i has at least one branching vertex (e.g., b_{i-h-1}) being added, and thus according to Lemma 4.2, it satisfies $\ell_{\mathcal{T}}(I_i) \leq \beta_k^{|I_i|}$. Consequently,

$$\begin{aligned} \ell_{\mathcal{T}}(I_0) &= \ell_{\mathcal{T}}(I_{h+1}) + \ell_{\mathcal{T}}(I_{h+2}) + \cdots + \ell_{\mathcal{T}}(I_{2h}) + \ell_{\mathcal{T}}(I_h) \\ &\leq \beta_k^{|I_{h+1}|} + \beta_k^{|I_{h+2}|} + \cdots + \beta_k^{|I_{2h}|} + \beta_k^{|I_h|} \\ &\leq \beta_k^{|I_0|-1} + \beta_k^{|I_0|-2} + \cdots + \beta_k^{|I_0|-h} + 1 \\ &\leq \frac{\beta_k^{|I_0|}-1}{\beta_k - 1} + 1 < 2 \cdot \beta_k^{|I_0|} \end{aligned}$$

The last inequality follows from the fact that $\beta_k > 1.5, \forall k \geq 1$. □

The time complexity of kDC then follows from Lemma 4.3, by noting that each node of the search tree (i.e., Lines 1–3 of Algorithm 1) takes $O(m)$ time.

Theorem 4.4. *Given a graph G and an integer k,* kDC *finds a maximum k-defective clique in $O(m \times \beta_k^n)$ time.*

Compared with the Analysis in [8]. We improve the base of the exponential time complexity of kDC [8] from γ_k to $\beta_k = \gamma_{k-1}$. This is achieved by using different analysis techniques for the nodes that already have branching vertices being added (i.e., Lemma 4.2) and for those that do not (i.e., Lemma 4.3); *this idea may also be beneficial to improving other problems.* Without this separation, we can only bound the length q of the path $(I_0, \ldots, I_q)$ by $k+1$ instead of k that is proved in Lemma 4.1. Also note that, Lemma 4.3 is non-inductive and has a coefficient 2 in the bound; if we use induction in the proof of Lemma 4.3, then the coefficient will become bigger and bigger and become exponential when going up the tree.

5 FURTHER IMPROVED TIME COMPLEXITIES WHEN $\omega_k(G)$ IS LARGE

The time complexity proved in Theorem 4.4, which has n in the exponent, holds for any graph. In this section, we show that the time complexity can be further improved for the graphs whose maximum k-defective cliques are large, by utilizing the diameter-two property of large k-defective cliques. We first bound the time complexity by using the degeneracy parameterization in the exponent in Section 5.1, and then bound the time complexity by using the degeneracy-gap parameterization in the exponent in Section 5.2.

5.1 Parameterize by the Degeneracy

Lemma 5.1 (Diameter-two Property of Large k-Defective Clique [12]). *For any k-defective clique, if it contains at least $k+2$ vertices, its diameter is at most two (i.e., any two non-adjacent vertices must have common neighbors in the defective clique).*

Following Lemma 5.1, if we know that $\omega_k(G) \geq k+2$, then for a backtracking instance (g, S) with $S \neq \emptyset$, we can remove from g the vertices whose shortest distance (computed in g) to any vertex of S is greater than two. This could significantly reduce the search space, as real graphs usually have a small average degree. However, it is difficult to utilize the diameter-two property reliably, since we do not know before-hand whether $\omega_k(G) \geq k+2$ or not and a k-defective clique of size smaller than $k+2$ may have a diameter larger than two. To resolve this, we propose to compute a maximum k-defective clique in two stages, where Stage-I utilizes the diameter-two property for pruning by assuming $\omega_k(G) \geq k+2$. If Stage-I fails (to find a k-defective clique of size at least $k+2$), then we go to Stage-II searching the graph again without utilizing the diameter-two property. This guarantees that a maximum k-defective clique is found regardless of its size.

Algorithm 2: kDC-two(G, k)

Input: A graph G and an integer k
Output: A maximum k-defective clique in G

```
1 C* ← ∅;
2 Let (v_1, ..., v_n) be a degeneracy ordering of the vertices of G;
3 for each v_i ∈ V(G) do
4     A ← N(v_i) ∩ {v_i, ..., v_n};
5     Let g_{v_i} be the subgraph of G induced by N[A] ∩ {v_i, ..., v_n};
6     Branch&Bound(g_{v_i}, {v_i});   /* Invoke Algorithm 1 */;
7 if |C*| < k + 1 then Branch&Bound(G, ∅);
8 return C*;
```

The pseudocode of our algorithm, denoted kDC-two, is shown in Algorithm 2; here, two refers to both "two"-stage and diameter-"two". We first compute a degeneracy ordering of the vertices of G (Line 2). Without loss of generality, let $(v_1, \ldots, v_n)$ be the degeneracy ordering, i.e., for each $1 \leq i \leq n$, v_i is the vertex with the smallest degree in the subgraph of G induced by $\{v_i, \ldots, v_n\}$; the degeneracy ordering can be computed in $O(m)$ time by the peeling algorithm [29]. Then, for each vertex $v_i \in V(G)$, we compute a largest diameter-two k-defective clique in which the first vertex, according to the degeneracy ordering, is v_i, by invoking the procedure Branch&Bound with input $(g_{v_i}, \{v_i\})$ (Lines 4–6); that is, the diameter-two k-defective clique contains v_i and is a subset of $(v_i, v_{i+1}, \ldots, v_n)$. Here, g_{v_i} is the subgraph of G induced by v_i and its neighbors and two-hop neighbors that come later than v_i according to the degeneracy ordering. After that, we check whether the currently found largest k-defective clique C^* is of size at least $k+1$: if $|C^*| \geq k+1$, then C^* is guaranteed to be a maximum k-defective clique of G; otherwise, $\omega_k(G) \leq k+1$ and a maximum k-defective clique of G may have a diameter larger than two. For the latter, we invoke Branch&Bound again with input $(G, \emptyset)$ (Line 7). Note that, we do not make use of the diameter-two property for pruning within the procedure Branch&Bound, and thus ensure the correctness of our algorithm.

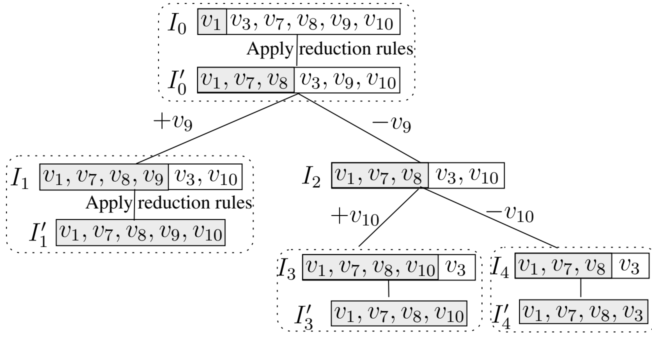

Figure 7: Running example

Example 5.2. Consider the graph G in Figure 3 and $k = 3$. A degeneracy ordering of G is $(v_1, v_2, \ldots, v_{14})$. When processing v_1, we only need to consider v_1, its neighbors $\{v_7, v_8\}$ and two-hop neighbors $\{v_3, v_9, v_{10}\}$; that is, we process the subgraph g_{v_1} of G induced by $\{v_1, v_3, v_7, \ldots, v_{10}\}$. The backtracking search tree for Branch&Bound$(g_{v_1}, \{v_1\})$, invoked at Line 6 of Algorithm 2 is shown in Figure 7, where vertices of S for each instance (g, S) are in the shaded area. For the root node I_0, applying reduction rule **RR2** adds v_7 and v_8 into S since each of them has at most one non-neighbor in the subgraph g_{v_1}; this results into the instance I'_0. Suppose the branching rule **BR** selects v_9 for I'_0 as v_9 is not adjacent to v_1. Then, two new instances I_1 and I_2 are generated as the two children of I'_0. For I_1, applying the reduction rule **RR1** removes v_3 from the graph and then applying **RR2** adds v_{10} into S; we reach a leaf node. Similarly, v_{10} is selected as the branching vertex for I_2, which then generates two leaf nodes I'_3 and I'_4.

Complexity Analysis of kDC-two**.** Following Theorem 4.4, Line 7 of Algorithm 2 runs in $O(m \times \beta_k^n)$ time. What remains is to bound $|I|$ for Line 6 of Algorithm 2. Let f denote the vertex obtained at Line 3 of Algorithm 2, g be the subgraph extracted at Line 5, and t be the number of f's non-neighbors in g. It holds that $d_g(f) = |A| \leq \alpha$ since a degeneracy ordering is used for extracting g, $t \leq \alpha(\Delta - 1)$, and $|V(g)| \leq d_g(f) + t + 1 \leq \alpha\Delta + 1$; here, α is the degeneracy of G and Δ is the maximum degree of G. Thus, kDC-two runs in $O(n \times |E(g)| \times \beta_k^{|V(g)|-1}) = O(n \times (\alpha\Delta)^2 \times \beta_k^{\alpha\Delta})$ time when $\omega_k(G) \geq k + 2$, and in $O(m \times \beta_k^n)$ time otherwise.

We show in the following that for the case of $\omega_k(G) \geq k + 2$, the exponent of the time complexity can be further reduced through a more refined analysis. Let $I_0 = (g_0, S_0)$ be the root of $\mathcal{T}$ for Line 6 of Algorithm 2; recall that, $f \in S_0$, $d_{g_0}(f) \leq d_g(f) \leq \alpha$ and $|I_0| \leq \alpha + t$. Let's consider the subtree of $\mathcal{T}$ formed by starting a depth-first-search from I_0 and backtracking once the path from I_0 to it has either t total edges or k positive edges (i.e., labeled as "+"); see the shaded subtree in Figure 2 for an illustration of $t = 4$ and $k = 2$. Let $\mathcal{L}$ be the set of leaf nodes of this subtree. Then, the number of leaf nodes of $\mathcal{T}$ satisfies

$$\ell_{\mathcal{T}}(I_0) \leq \textstyle\sum_{I \in \mathcal{L}} \ell_{\mathcal{T}}(I)$$

We bound $|\mathcal{L}|$ and $\ell_{\mathcal{T}}(I)$ for $I \in \mathcal{L}$ in the following two lemmas.

Lemma 5.3. *$|\mathcal{L}| = O(t^k)$ where t is the number of f's non-neighbors in g.*

Proof. To bound $|\mathcal{L}|$, we observe that the search tree $\mathcal{T}$ is a full binary tree with each node having a positive edge to its left child and a negative edge to its right child. Thus, we can label every edge by its level in the tree (see Figure 2), and for each node $I \in \mathcal{L}$, we associate with it a set of numbers corresponding to the levels of the positive edges from I_0 to I. Then, it is easy to see that each node of $\mathcal{L}$ is associated with a distinct subset, of size at most k, of $\{1, 2, \ldots, t\}$. Consequently, $|\mathcal{L}| \leq \sum_{i=0}^{k} \binom{t}{i} = O(t^k)$ [30]. □

Lemma 5.4. *For all $I \in \mathcal{L}$, it holds that $|I| \leq \alpha$ and $|V(I.g)| \leq \alpha + k + 1$.*

Proof. Let's consider the path $(I_0, I_1, \ldots, I_{p-1}, I_p = I)$ from I_0 to I. If there are k positive edges on the path, then all vertices of $V(g_p) \setminus S_p$ must be adjacent to f, and thus $|I_p| \leq \alpha$ and $|V(g_p)| \leq \alpha + k + 1$. The latter holds since all of f's non-neighbors in g are in S and thus of quantity at most k. The former can be shown by contradiction. Suppose $V(g_p) \setminus S_p$ contains a non-neighbor of f, then adding each of the k branching vertices (on the positive edges of the path) must bring at least one non-edge to S_p, due to the branching rule **BR**; then **RR1** will remove all non-neighbors of f from $V(g_p) \setminus S_p$, contradiction.

Otherwise, there are at most $k - 1$ positive edges on the path and $p = t$. Then, $|I_p| \leq |I_0| - t \leq \alpha$. Also, there are at least $t - k + 1$ negative edges on the path and thus $|V(g_p)| \leq |V(g_0)| - (t-k+1) \leq t + \alpha + 1 - (t - k + 1) = \alpha + k$. □

Consequently, the number of leaf nodes of $\mathcal{T}$ is $O((\alpha\Delta)^k \beta_k^{\alpha})$ and the time complexity of kDC-two follows.

Theorem 5.5. *Given a graph G and an integer k,* kDC-two *runs in $O(n \times (\alpha\Delta)^{k+2} \times \beta_k^{\alpha})$ time when $\omega_k(G) \geq k+2$, and in $O(m \times \beta_k^n)$ time otherwise.*

We represent the graph in CSR (Compressed Sparse Row) format in main memory [10], and dynamically rearrange the vertex's neighbors for the subgraph g during the recursion of Branch&Bound (Algorithm 1) in a similar way as [15]. As a result, the space complexity of kDC-two is $O(n + m)$.

Remark. Although the idea of using the diameter-two property to reduce the search space is not new, it has not been incorporated into the existing practical algorithms for maximum k-defective clique computation, such as MADEC$^+$ [12], KDBB [18], kDC [8] and KD-Club [24]. We show how to exploit the diameter-two-based pruning for maximum k-defective clique computation while ensuring the algorithm's correctness, i.e., successfully find a maximum k-defective clique even if it is smaller than $k + 2$; recall that, we do not know whether $\omega_k(G) \geq k + 2$ or not before running the algorithm. We also analyzed the time complexity of the algorithm and will show its effectiveness in practice in Section 7.

Although we focus on sequential algorithms in this paper, kDC-two can be easily parallelized as follows. Firstly, we can parallelize the for loop at Line 3 of Algorithm 2. For each vertex v_i, we extract the subgraph g_{v_i} and then process g_{v_i}; the tasks generated for different vertices are independent from each other and thus can be run in parallel. Similarly, we can parallelize Line 7 by unrolling the first level of the recursion tree, e.g., processing the "children" $I_{h+1}, \ldots, I_{2h}$ in parallel for node I_0 in Figure 6.

5.2 Parameterize by the Degeneracy Gap

In this subsection, we prove that when $\omega_k(G) \geq k+2$, a maximum k-defective clique can also be found in $O^*((\alpha\Delta)^{k+2}(k+1)^{\alpha+k+1-\omega_k(G)})$ time, by using the degeneracy-gap parameterization $\alpha+k+1-\omega_k(G)$ which is at most $\alpha-1$. This is better than $O^*((\alpha\Delta)^{k+2}\beta_k^{\alpha})$ when $\omega_k(G)$ is close to $\alpha+k+1$, the degeneracy-based upper bound of $\omega_k(G)$, i.e., $\omega_k(G) \leq \alpha+k+1$ always holds.

Let's consider the problem of testing whether G has a k-defective clique of size τ. To do so, we truncate the search tree $\mathcal{T}$ by cutting the entire subtree rooted at node I if $|V(I.g)| \leq \tau$; that is, we terminate Branch&Bound once $|V(g')| \leq \tau$ after Line 2 of Algorithm 1. Let $\mathcal{T}'$ be the truncated version of $\mathcal{T}$. We first bound the number of leaf nodes of $\mathcal{T}'$ in the following two lemmas, in a similar fashion as Lemmas 4.2 and 4.3; proofs are omitted due to limit of space.

Lemma 5.6. *For any node I of $\mathcal{T}'$ that has at least one branching vertex being added, it holds that $\ell_{\mathcal{T}'}(I) \leq (k+1)^{|V(I.g)|-\tau}$.*

Lemma 5.7. *For any node I of $\mathcal{T}'$, it holds that $\ell_{\mathcal{T}'}(I) < 2\cdot(k+1)^{|V(I.g)|-\tau}$.*

Then, the following time complexity can be proved in a similar way to Theorem 5.5 but using Lemma 5.7 and Lemma 5.4.

Lemma 5.8. *Testing whether G has a k-defective clique of size τ for $\tau \geq k+2$ takes $O(n \times (\alpha\Delta)^{k+2} \times (k+1)^{\alpha+k+1-\omega_k(G)})$ time.*

Finally, we can find a maximum k-defective clique by iteratively testing whether G has a k-defective clique of size τ for $\tau = \{\alpha+k+1, \alpha+k, \ldots\}$. This will find the maximum k-defective clique and terminate after testing $\tau = \omega_k(G)$. Consequently, the following time complexity follows.

Theorem 5.9. *A maximum k-defective clique in G can be found in $O((\alpha+k+2-\omega_k(G)) \times n \times (\alpha\Delta)^{k+2} \times (k+1)^{\alpha+k+1-\omega_k(G)})$ time when $\omega_k(G) \geq k+2$.*

Remark. We note that a similar time complexity has been proven in [46]. Specifically, it shows that a maximum k-defective clique in a graph G with $\omega_k(G) \geq k+2$ can be found in $O^*((2k+2)^{\alpha_2+1-\omega_k(G)})$ time, where $\alpha_2 \geq \alpha$ is a quantity similar to $\alpha\Delta$ that bounds (and minimizes) the maximum vertex number among the subgraphs extracted at Line 5 of Algorithm 2. By using our techniques above, we can improve the time complexity of [46] to $O^*((k+1)^{\alpha_2+1-\omega_k(G)})$, reducing the base from $2k+2$ to $k+1$. Note however that α_2 could be much larger than $\alpha+k$. We omit the details.

6 A NEW REDUCTION RULE

In this section, we propose a new reduction rule based on the degree-sequence-based upper bound **UB** that is proposed in [18], to further improve the practical performance of our algorithm.

> **UB [18].** 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 $\sum_{j=1}^{i} |\overline{N}_S(v_j)| \leq k - |\overline{E}(S)|$.

Note that, different tie-breaking techniques for ordering the vertices lead to the same upper bound. Thus, an arbitrary tie-breaking technique can be used in **UB**.

Let lb be the size of the currently found best solution. If an upper bound computed by **UB** for an instance (g, S) is no larger than lb, then we can prune the instance. However, this way of first generating an instance and then try to prune it based on a computed upper bound is inefficient. To improve efficiency, we propose to remove $u \in V(g)\setminus S$ from g if an upper bound of $(g, S \cup u)$ is no larger than lb. Note that, rather than computing the exact upper bound for $(g, S \cup u)$, we only need to test whether the upper bound is larger than lb or not. The latter can be conducted more efficiently and without generating $(g, S \cup u)$; moreover, computation can be shared between the testing for different vertices of $V(g) \setminus S$. *We remark that this general idea of turning an upper bound into a reduction rule may also be beneficial to other problems.*

Let $v_1, v_2, \ldots$ be an ordering of $V(g) \setminus (S \cup u)$ in non-decreasing order regarding $|\overline{N}_S(\cdot)|$, and C be the set of vertices that have the same number of non-neighbors in S as $v_{lb-|S|}$, i.e., $C = \{v_i \in V(g) \setminus (S \cup u) \mid |\overline{N}_S(v_i)| = |\overline{N}_S(v_{lb-|S|})|\}$. Let C_1 and C_2 be a partitioning of C according to their positions in the ordering, i.e., $C_1 = C \cap \{v_1, \ldots, v_{lb-|S|}\}$ and $C_2 = C \setminus C_1$. Note that, both C_1 and C_2 contain consecutive vertices in the ordering, and C_2 could be empty. A visualization of the ordering and vertex sets is shown below, where $S \cup u$ and $lb - |S|$ are denoted by R and r for brevity.

$$\underbrace{S, u}_{R},\ \underbrace{v_1, \ldots, v_{r-|C_1|}}_{D},\ \overbrace{\underbrace{v_{r-|C_1|+1}, \ldots, v_r}_{C_1},\ \underbrace{v_{r+1}, \ldots v_{r+|C_2|}}_{C_2}}^{C = \{v_i \in V(g) \setminus R \mid |\overline{N}_S(v_i)| = |\overline{N}_S(v_r)|\}},\ \ldots$$

We prove in the lemma below that the upper bound computed by **UB** for the instance $(g, S \cup u)$ is at most lb if and only if

$$|\overline{E}(S)| + \sum_{j=1}^{r} |\overline{N}_S(v_j)| + |\overline{N}_{S\cup D}(u)| + \max\{|\overline{N}_{C_1}(u)| - |N_{C_2}(u)|, 0\} > k \tag{2}$$

Lemma 6.1. *The upper bound computed by* **UB** *for the instance $(g, S \cup u)$ is at most lb if and only if Equation (2) is satisfied.*

Based on the above discussions, we propose the following degree-sequence-based reduction rule **RR3**.

> **RR3 (degree-sequence-based reduction rule).** Given an instance (g, S) with $|S| < lb < |V(g)|$ and a vertex $u \in V(g) \setminus S$, we remove u from g if Equation (2) is satisfied.

Algorithm 3: ApplyRR3(g, S, lb)

1 Obtain $|\overline{E}(S)|$ and $|\overline{N}_S(v)|$ for each $v \in V(g) \setminus S$;
2 Let $u_1, u_2, \ldots$ be an ordering of $V(g) \setminus S$ in non-decreasing order regarding $|\overline{N}_S(\cdot)|$;
3 $X \leftarrow \emptyset$;
4 **for** $i \leftarrow 1$ ***to*** $|V(g) \setminus S|$ **do**
5   **if** $|X| + |V(g) \setminus S| - i < lb - |S|$ **then return** $(g[S], S)$;
6   Let $v_1, v_2, \ldots$ be the vertices of $X \cup \{u_{i+1}, u_{i+2}, \ldots\}$ in non-decreasing order regarding $|\overline{N}_S(\cdot)|$;
7   Obtain $|\overline{N}_D(u_i)|$, $|\overline{N}_{C_1}(u_i)|$ and $|N_{C_2}(u_i)|$;
8   **if** *Equation (2) is not satisfied* **then** Append u_i to the end of X;
9 **return** $(g[S \cup X], S)$;

Given an instance (g, S), our pseudocode of efficiently applying **RR3** for all vertices of $V(g) \setminus S$ is shown in Algorithm 3, which returns a reduced instance at either Line 5 or Line 9. We first obtain $|\overline{E}(S)|$ and $|\overline{N}_S(v)|$ for each $v \in V(g) \setminus S$ (Line 1), and then order vertices of $V(g) \setminus S$ in non-decreasing order regarding $|\overline{N}_S(\cdot)|$ (Line 2). Let $u_1, u_2, \ldots$ be the ordered vertices. We then process the vertices of $V(g) \setminus S$ one-by-one according to the sorted order (Line 4). When processing u_i, the vertices that we need to consider are the vertices that have not been processed yet (i.e., $\{u_{i+1}, u_{i+2}, \ldots\}$) and the subset of $\{u_1, u_2, \ldots, u_{i-1}\}$ that passed (i.e., not pruned by) the reduction rule; denote the latter subset by X. This means that $\{u_1, \ldots, u_{i-1}\} \setminus X$ have already been removed from g by the reduction rule. If the number of remaining vertices (i.e., $|X| + |V(g) \setminus S| - i$) is less than $r = lb - |S|$, then we remove all vertices of $V(g) \setminus S$ from g by returning $(g[S], S)$ (Line 5). Otherwise, let $v_1, v_2, \ldots, v_r, \ldots$ be the vertices of $X \cup \{u_{i+1}, u_{i+2}, \ldots\}$ in non-decreasing order regarding $|\overline{N}_S(\cdot)|$ (Line 6). We obtain $|\overline{N}_D(u_i)|$, $|\overline{N}_{C_1}(u_i)|$ and $|N_{C_2}(u_i)|$ (Line 7), and remove u_i from g if Equation (2) is satisfied; otherwise, u_i is not pruned and is appended to the end of X (Line 8). Finally, Line 9 returns the reduced instance $(g[S \cup X], S)$. From the pseudocode, it is easy to see that our new reduction rule **RR3** can be used in any branch-and-bound algorithm for maximum k-defective clique computation, since it simply removes non-promising vertices from $V(g) \setminus S$ based on a given lower bound lb.

Lemma 6.2. *Algorithm 3 runs in $O(|E(g)| + k)$ time.*

Proof. Firstly, Lines 1–2 run in $O(|E(g)| + k)$ time, where the sorting is conducted by counting sort. Secondly, Line 6 does not do anything; it is just a syntax sugar for relabeling the vertices. Thirdly, Line 7 can be conducted in $O(|N_g(u_i)|)$ time since $|\overline{N}_D(u_i)| = |D| - |N_D(u_i)|$ and $|\overline{N}_{C_1}(u_i)| = |C_1| - |N_{C_1}(u_i)|$; note that, as each of D, C_1, C_2 spans at most two arrays (i.e., X and $\{u_{i+1}, \ldots\}$), we can easily get its size and boundary. Lastly, Line 8 can be checked in constant time by noting that $\sum_{j=1}^{r} |\overline{N}_S(v_j)|$ can be obtained in constant time after storing the suffix sums of $|\overline{N}_S(u_1)|, |\overline{N}_S(u_2)|, \ldots, |\overline{N}_S(u_{i+1})|, |\overline{N}_S(u_{i+2})|, \ldots$. □

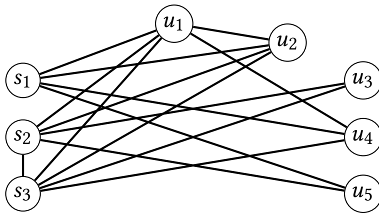

Figure 8: Example instance for applying reduction rule RR3

Example 6.3. Consider the instance (g, S) in Figure 8 for $k = 3$ and $lb = 5$, where g is the entire graph and $S = \{s_1, s_2, s_3\}$; thus $r = 2$. The values of $|\overline{N}_S(\cdot)|$ for the vertices of $V(g) \setminus S = \{u_1, \ldots, u_5\}$ are $\{u_1 : 0, u_2 : 0, u_3 : 1, u_4 : 1, u_5 : 1\}$. As $|\overline{E}(S)| = 2$, the upper bound of (g, S) computed by **UB** is 6; thus, the instance is not pruned.

Let's apply **RR3** for u_1. As $X = \emptyset$, we have $v_i = u_{i+1}$ for $1 \leq i \leq 4$. Then $D = \{u_2\}, C_1 = \{u_3\}, C_2 = \{u_4, u_5\}$, $\sum_{j=1}^{r} |\overline{N}_S(v_j)| = 1$, $|\overline{N}_{S \cup D}(u_1)| = 0$, $|\overline{N}_{C_1}(u_1)| = 1$ and $|N_{C_2}(u_1)| = 1$. Thus, Equation (2) is not satisfied; u_1 is not pruned and is appended to X.

Now let's apply **RR3** for u_2. As $X = \{u_1\}$, we have $v_1 = u_1$ and $v_i = u_{i+1}$ for $2 \leq i \leq 4$. Then $D = \{u_1\}, C_1 = \{u_3\}, C_2 = \{u_4, u_5\}$, $\sum_{j=1}^{r} |\overline{N}_S(v_j)| = 1$, $|\overline{N}_{S \cup D}(u_2)| = 0$, $|\overline{N}_{C_1}(u_2)| = 1$ and $|N_{C_2}(u_2)| = 0$. Consequently, Equation (2) is satisfied and u_2 is removed from g. It can be verified that u_3, u_4, u_5 will all subsequently be removed.

Effectiveness of RR3. kDC [8] also proposed a reduction rule based on **UB**; let's denote it as **RR3'**. We remark that our **RR3** is more effective (i.e., prunes more vertices) than **RR3'** since the latter ignores the non-edges between u and $V(g) \setminus (S \cup u)$ that are considered by **RR3**; specifically, **RR3'** removes u from g if $|\overline{E}(S)| + \sum_{j=1}^{r} |\overline{N}_S(v_j)| + |\overline{N}_S(u)| > k$. More generally, the effectiveness of our **RR3** is characterized by the lemma below.

Lemma 6.4. **RR3** *is more effective than any other reduction rule that is designed based on an upper bound of $(g, S \cup u)$ that ignores all the non-edges between vertices of $V(g) \setminus (S \cup u)$.*

Proof. Firstly, we have proved in Lemma 6.1 that applying **RR3** is equivalent to computing **UB** for $(g, S \cup u)$. Secondly, it can be shown that **UB** computes the tightest upper bound for $(g, S \cup u)$ among all upper bounds that ignore all the non-edges between vertices of $V(g) \setminus (S \cup u)$. Thus, the lemma holds. □

In particular, the second-order reduction rule proposed in [8] is designed based on an upper bound of $(g, S \cup u)$ that does not consider the non-edges between vertices of $V(g) \setminus (S \cup u)$. Thus, **RR3** is more effective than the second-order reduction rule of [8].

7 EXPERIMENTS

In this section, we evaluate the efficiency of our techniques, by comparing the following algorithms. [1]

- kDC-two: our algorithm as shown in Algorithm 2.
- kDC2+RR3: the variant of kDC-two that also incorporates our new reduction rule **RR3** presented in Section 6.
- kDC [2]: the existing algorithm proposed in [8].
- KD-Club [3]: the existing algorithm proposed in [24].

All algorithms are implemented in C++ and compiled with the -O3 flag. All experiments are conducted in single-thread modes on a machine with an Intel Core i7-8700 CPU and 64GB main memory.

We run the algorithms on the following three graph collections, which are the same ones tested in [8, 18].

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

[1] Note that, the effectiveness of the defective clique model in serving applications has been demonstrated in [14, 49]. Thus, we do not repeat the effectiveness testing here.
[2] https://lijunchang.github.io/Maximum-kDC/
[3] https://github.com/JHL-HUST/KD-Club
[4] http://lcs.ios.ac.cn/~caisw/Resource/realworld%20graphs.tar.gz
[5] https://networkrepository.com/socfb.php
[6] https://www.cc.gatech.edu/dimacs10/downloads.shtml
[7] http://snap.stanford.edu/data/

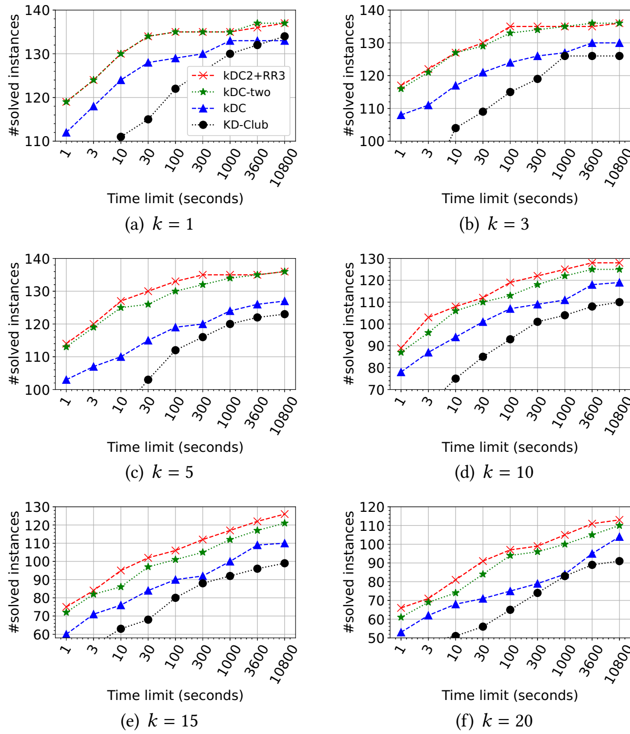

Figure 9: Number of solved instances by varying time limit for real-world graphs collection (best viewed in color)

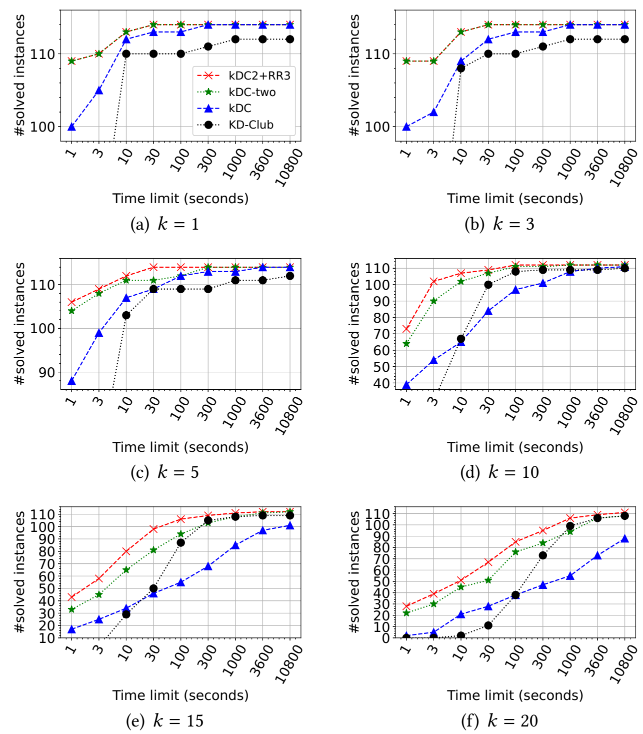

Figure 10: Number of solved instances by varying time limit for Facebook graphs collection (best viewed in color)

Table 2: Number of instances solved by kDC2+RR3 (abbreviated as kDC2+R), kDC and KD-Club with a time limit of 3 hours (best performers are highlighted in bold)

	Real-world collection			Facebook collection			DIMACS10&SNAP		
	kDC2+R	kDC	KD-Club	kDC2+R	kDC	KD-Club	kDC2+R	kDC	KD-Club
$k = 1$	**137**	133	134	**114**	**114**	112	**37**	**37**	**37**
$k = 3$	**136**	130	126	**114**	**114**	112	**37**	**37**	36
$k = 5$	**136**	127	123	**114**	**114**	112	**37**	**37**	36
$k = 10$	**128**	119	110	**112**	111	110	**36**	**36**	32
$k = 15$	**126**	110	99	**112**	101	109	**36**	29	30
$k = 20$	**113**	104	91	**111**	88	108	**34**	27	26

Same as [8, 18], we choose k from $\{1, 3, 5, 10, 15, 20\}$. We set a time limit of 3 hours for each individual testing (i.e., running a particular algorithm on a specific graph instance with a chosen k value); thus, the experimental results are not affected by the processing order of the graph instances.

7.1 Against the Existing Algorithms

In this subsection, we evaluate the practical efficiency of our algorithm kDC2+RR3 against the two most recent algorithms kDC and KD-Club. We do not include other existing algorithms such as MADEC$^+$ and KDBB since they have shown to be outperformed by kDC and KD-Club in [8] and [24], respectively; however, there is no direct comparison between kDC and KD-Club in the literature. We also do not include the algorithm of [46], since it is of theoretical interest only and no implementation is available.

We first report in Table 2 the total number of graph instances that are solved by each algorithm with a time limit of 3 hours for each graph instance. We can see that for all three algorithms, the number of solved instances decreases when k increases; this indicates that *when k increases, the problem becomes more difficult and takes more time to solve*. Nevertheless, our algorithm kDC2+RR3 consistently outperforms the two existing algorithms by solving more instances within the time limit. The improvement is more profound when k becomes larger. For example, for $k = 15$, kDC2+RR3 solves 16, 11 and 7 (resp. 27, 3 and 6) more instances than kDC (resp. KD-Club) on the three graph collections, respectively.

Secondly, we compare the number of instances solved by the algorithms when varying the time limit from 1 second to 3 hours for each graph instance. The results on the real-world collection and Facebook collection for different k values are shown in Figures 9 and 10, respectively; note that, to make the difference between kDC2+RR3 and kDC-two more visible, we truncated the results of KD-Club for small time limits. We can see that our algorithm kDC2+RR3 consistently outperforms kDC and KD-Club across all the time limits. In particular, our algorithm kDC2+RR3 solves all 114 Facebook graphs with a time limit of 30 seconds for $k = 1$, 3 and 5, while the time limits needed by kDC are 125, 393 and 1353 seconds, respectively; on the other hand, KD-Club is not able to solve all instances with a time limit of 3 hours. However, there is no clear winner between kDC and KD-Club. kDC generally performs better, while KD-Club outperforms kDC on Facebook graphs for $k \geq 10$ and with large time limits. This is because KD-Club conducts more aggressive and time-consuming pruning which is effective in these cases, but does not pay off in other cases.

Table 3: Running time and memory usage of kDC2+RR3 (abbreviated as kDC2+R), kDC-two, kDC+R, kDC, and kDC-Club on the 33 graphs with more than 100,000 vertices from the real-world graphs collection. Best performers are highlighted in bold: specifically, if the running time is slower than the fastest time by only less than 10%, it is considered as the best.

		Running time (in seconds)										Memory usage (in MB)				
		$k = 10$					$k = 15$					$k = 10$				
	m	kDC2+R	kDC-two	kDC+R	kDC	KD-Club	kDC2+R	kDC-two	kDC+R	kDC	KD-Club	kDC2+R	kDC-two	kDC+R	kDC	KD-Club
ca-citeseer	814K	**0.02**	**0.02**	**0.02**	0.02	0.05	**0.02**	**0.02**	**0.02**	0.03	0.05	16	16	16	16	42
ca-coauthors-dblp	15M	**0.20**	**0.21**	0.25	0.25	14	**0.22**	**0.22**	0.34	0.37	40	137	137	137	137	604
ca-dblp-2010	716K	**0.02**	**0.02**	**0.02**	**0.02**	0.06	**0.02**	**0.02**	0.02	0.03	0.10	15	15	15	15	38
ca-dblp-2012	1M	**0.04**	**0.03**	**0.03**	0.05	0.17	**0.04**	**0.03**	0.04	0.05	0.32	21	21	21	21	54
ca-hollywood-2009	56M	**0.79**	**0.74**	**0.74**	0.83	130	**0.78**	**0.75**	**0.75**	1.0	132	466	466	466	466	2209
ca-MathSciNet	820K	**0.03**	**0.03**	0.04	0.06	0.74	**0.04**	**0.04**	0.13	0.13	12	20	20	20	20	47
rt-retweet-crawl	2M	**0.22**	**0.22**	**0.22**	0.26	-	251	**218**	5884	-	-	67	67	67	67	-
sc-ldoor	20M	**1779**	2195	-	-	-	**3890**	5082	-	-	-	560	560	-	-	-
sc-msdoor	9M	**937**	1165	-	-	-	**2098**	2845	-	-	-	254	254	-	-	-
sc-pwtk	5M	**19**	21	-	-	-	**1409**	3843	-	-	-	154	154	-	-	-
sc-shipsec1	1M	**0.15**	0.19	17	17	3099	**0.62**	0.89	114	117	-	33	32	32	35	466
sc-shipsec5	2M	**0.24**	**0.26**	29	29	8003	**5.9**	11	448	471	-	43	43	43	46	1130
soc-delicious	1M	**0.09**	0.11	0.53	1.1	78	**0.33**	0.49	4.5	13	3878	34	34	34	34	74
soc-digg	5M	**152**	266	3092	-	**167**	7499	-	-	-	**1220**	161	166	161	-	258
soc-douban	327K	**0.03**	**0.03**	**0.03**	0.03	-	**48**	70	-	-	-	16	15	16	15	-
soc-flixster	7M	**32**	68	843	1713	289	**980**	2714	-	-	10478	158	158	158	158	403
soc-gowalla	950K	**1.00**	3.3	7.3	23	5.2	**54**	659	226	2549	291	20	20	20	20	47
soc-lastfm	4M	2440	-	**1014**	-	-	-	-	-	-	-	103	-	103	-	-
soc-livejournal	27M	**2.1**	**2.1**	**2.2**	**2.3**	37	**2.1**	**2.1**	**2.1**	2.7	37	343	343	343	343	1207
soc-LiveMocha	2M	**2461**	-	3472	-	-	-	-	-	-	-	68	-	68	-	-
soc-orkut	106M	**505**	1140	-	-	6091	**7151**	-	-	-	-	2129	2131	-	-	4506
soc-pokec	22M	**5.8**	**5.8**	**5.9**	7.1	1580	**9.9**	12	19	33	-	401	401	401	401	936
soc-youtube	1M	**26**	414	72	1143	-	**623**	-	939	-	-	47	47	47	47	-
soc-youtube-snap	2M	**46**	920	162	3245	-	**1111**	-	1830	-	-	75	75	75	75	-
socfb-A-anon	23M	31	66	**22**	78	-	240	627	**159**	1167	-	432	432	432	433	-
socfb-B-anon	20M	50	79	**21**	49	-	1449	2778	**572**	2277	-	492	492	492	493	-
tech-as-skitter	11M	**0.67**	0.76	**0.67**	0.77	34	**1.6**	2.9	3.4	17	71	145	145	145	145	485
tech-RL-caida	607K	**0.12**	0.17	0.59	0.90	378	**1.3**	2.4	10.0	22	-	22	21	22	21	40
web-arabic-2005	1M	**0.02**	**0.02**	**0.02**	**0.02**	0.12	**0.02**	**0.02**	0.02	0.02	0.20	21	21	21	21	75
web-it-2004	7M	**0.17**	**0.17**	0.38	0.37	201	**0.18**	**0.20**	0.58	0.55	205	76	76	76	76	309
web-sk-2005	334K	**0.01**	**0.01**	0.01	0.01	0.48	**0.01**	**0.01**	0.02	0.02	0.51	9	9	9	9	20
web-uk-2005	11M	**0.23**	**0.24**	0.61	0.59	353	**0.25**	**0.26**	0.87	0.83	358	100	100	100	100	475
web-wikipedia2009	4M	**1.00**	**0.99**	**1.00**	**1.0**	11	**1.0**	**0.99**	**1.1**	1.3	20	166	159	166	159	239

Table 4: Average speed-ups of kDC2+RR3 (abbreviated as kDC2+R) against kDC-two, kDC, and KD-Club for different graph types in the real-world graphs collection

	$k = 10$			$k = 15$		
	kDC-two / kDC2+R	kDC / kDC2+R	KD-Club / kDC2+R	kDC-two / kDC2+R	kDC / kDC2+R	KD-Club / kDC2+R
Biological Networks	1.77×	3.91×	84.64×	3.14×	3.35×	11.57×
Collaboration Networks	1.14×	1.66×	48.94×	1.00×	1.93×	50.75×
Interaction Networks	2.74×	8.57×	49.81×	2.63×	4.48×	20.83×
Infrastructure Networks	1.22×	1.24×	59.39×	2.46×	2.64×	22.66×
Retweet Networks	1.21×	1.25×	369.61×	1.09×	4.15×	178.41×
Scientific Computing	1.15×	70.04×	925.05×	1.71×	41.64×	187.66×
Temporal Reachability	1.08×	1.48×	9.04×	1.17×	2.21×	8.59×
Social Networks	2.77×	8.95×	25.40×	2.78×	9.08×	17.81×
Facebook Networks	1.65×	7.00×	20.44×	2.14×	19.88×	15.20×
Technological Networks	1.68×	6.39×	44.28×	1.78×	10.18×	908.14×
Web Graphs	1.05×	2.03×	23.78×	1.25×	3.23×	29.19×

Thirdly, to evaluate the performance of our algorithm on different graph types, we group the graphs in the real-world graphs collection into 11 types according to https://networkrepository.com, and compute the average (i.e., geometric mean) speed-ups of kDC2+RR3 over kDC and KD-Club for each graph type. The results are reported in Table 4. We can see that kDC2+RR3 outperforms the existing algorithms across all different graph types, though speed-ups vary.

Fourthly, we report the running time of kDC2+RR3, kDC and KD-Club on graphs with more than 100,000 vertices from the real-world graphs collection for $k = 10$ and 15 in Table 3, where '–' indicates that the running time is longer than the 3-hour limit. There are totally 33 such graphs; note that, we ignored the graphs that none of the algorithms can finish within 3 hours. The number of edges for each graph is given in the second column. From Table 3, we can observe that our algorithm kDC2+RR3 consistently outperforms kDC, which in turn runs significantly faster than KD-Club. The only exception is on "soc-digg" and for $k = 15$, where KD-Club runs the fastest; this is because the aggressive pruning of KD-Club in the preprocessing stage reduces the input graph to a subgraph of 749 vertices in 294 seconds, while that of kDC2+RR3 reduces the graph to a subgraph of 4177 vertices in 2 seconds. On average (geometric mean), kDC2+RR3 is 13× and 9× (resp. 179× and 69×) faster than kDC (resp. KD-Club) for $k = 10$ and $k = 15$, respectively; in this calculation, we ignored graphs that are easy to solve (i.e., solved by both algorithms within 1 second), and treat '-' as 3 hours. Memory usages of the algorithms for $k = 10$ are also shown in Table 3; the results for $k = 15$ are similar and are omitted. We can see that our algorithm kDC2+RR3 consumes similar memory as kDC, and much less than KD-Club; we remark that the latter is mainly due to implementation differences, and our algorithm kDC2+RR3 is implemented based on the code base of kDC.

In summary, our algorithm kDC2+RR3 consistently solves more graph instances than the most recent algorithms kDC and KD-Club when varying the time limit from 1 second to 3 hours, and also consistently runs faster than them across the different graphs with an average speed up up-to two orders of magnitude.

7.2 Ablation Studies

In this subsection, we first evaluate the effectiveness of our new reduction rule **RR3** by comparing kDC2+RR3 with kDC-two; recall that the only difference between kDC2+RR3 and kDC-two is that kDC2+RR3 incorporates the reduction rule **RR3**. The results of kDC-two are also shown in Figures 9 and 10 and Table 3. We can observe from Figures 9 and 10 that the reduction rule **RR3** enables kDC2+RR3 to solve more graph instances across the different time limits than kDC-two. In particular, kDC2+RR3 solves 3, 5 and 3 more instances than kDC-two for $k = 10$, 15 and 20, respectively, on the real-world graphs collection with the time limit of 3 hours. From Table 3, we can see that kDC2+RR3 consistently runs faster than kDC-two across the different graphs with an average (geometric mean) speed up of 2.5× and 2.2× for $k = 10$ and $k = 15$, respectively. We also run the variant kDC+R, which is kDC equipped with the reduction rule **RR3**, and report its running time in Table 3. kDC+R consistently runs faster than kDC. This demonstrates the practical effectiveness of our new reduction rule **RR3**.

Table 5: Number of graphs with small maximum k-defective clique (i.e., $\omega_k(G) \leq k+1$) and number of graphs with large maximum k-defective clique (i.e., $\omega_k(G) \geq k+2$)

	Real-world collection		Facebook collection		DIMACS10&SNAP	
	#small	#large	#small	#large	#small	#large
$k = 1$	2	137	0	114	0	37
$k = 3$	13	126	0	114	0	37
$k = 5$	22	117	1	113	1	36
$k = 10$	40	98	1	111	8	29
$k = 15$	47	91	1	111	12	25
$k = 20$	53	83	1	111	16	21

Secondly, we compare kDC-two with kDC. Note that, the only difference between them is that kDC-two conducts the computation in two stages and exploits the diameter-two property for pruning in Stage-I. From Figures 9 and 10 and Table 3, we can see that kDC-two consistently outperforms kDC. Specifically, kDC-two solves 6, 11 and 6 more instances than kDC for $k = 10$, 15 and 20, respectively, on the real-world graphs collection with the time limit of 3 hours. In Table 3, kDC-two is on average (geometric mean) 7.9× and 6.3× faster than kDC for $k = 10$ and $k = 15$, respectively. From Table 3, we can also see that kDC2+RR3 generally runs faster than kDC+R; the latter runs faster on "socfb-B-anon" mainly due to the different branching rule that is implicitly used at Line 3 of Algorithm 2. This demonstrates the practical effectiveness of diameter-two-based pruning. To gain more insights, we report in Table 5 the number of graphs with small maximum k-defective clique (i.e., $\omega_k(G) \leq k+1$) and the number of graphs with large maximum k-defective clique (i.e., $\omega_k(G) \geq k+2$) for each graph collection and k value. We can see that when k increases, the proportion of graphs with $\omega_k(G) \geq k+2$ decreases. Nevertheless, even for $k = 20$, there are still a lot of graphs with $\omega_k(G) \geq k+2$ such that kDC-two runs in $O^*((\alpha\Delta)^{k+2}\gamma_{k-1}^{\alpha})$ time; this is especially true for Facebook graphs.

Thirdly, we report in Table 6 the number of graphs where the time complexity proved in Section 5.2 is better than Section 5.1, i.e., $(\alpha + k + 2 - \omega_k(G)) \times (k+1)^{\alpha+k+1-\omega_k(G)} \leq \gamma_{k-1}^{\alpha}$. We can see that for $k \geq 10$, there are very few graphs where the time complexity of Section 5.2 is better. This is because the gap between the upper bound $\alpha + k + 1$ and $\omega_k(G)$ is large in these cases. This suggests that the degeneracy gap in practice is typically large for large k and thus the algorithm of Section 5.2 is only of theoretical interest.

Table 6: #graphs s.t. $(\alpha{+}k{+}2{-}\omega_k(G)){\times}(k{+}1)^{\alpha+k+1-\omega_k(G)} \leq \gamma_{k-1}^{\alpha}$

	$k = 1$	$k = 3$	$k = 5$	$k = 10$	$k = 15$	$k = 20$
Real-world graphs collection	116	69	45	24	19	15
Facebook graphs collection	100	55	32	4	0	0
DIMACS10&SNAP	33	22	17	7	4	3

Finally, we remark that when k increases from 1 to 20, the maximum k-defective clique size increases by 32% on average for the real-world graphs. As discussed earlier, the running time increases as well. Thus, choosing an appropriate k is application dependent.

8 RELATED WORK

The concept of defective clique was formulated in [49]. Early algorithms for maximum k-defective clique computation, such as those proposed in [19, 20, 45], are inefficient and can only deal with small graphs. The MADEC$^+$ algorithm [12] is the first algorithm that can handle large graphs. The KDBB algorithm [18] improves the practical performance by proposing preprocessing as well as multiple pruning techniques. kDC [8] and KD-Club [24] are the two most recent algorithms and are included in our empirical study. From the theoretical perspective, among the existing algorithms, only MADEC$^+$ [12] and kDC [8] beats the trivial time complexity of $O^*(2^n)$. Specifically, MADEC$^+$ runs in $O^*(\gamma_{2k}^n)$ time while kDC improves the time complexity to $O^*(\gamma_k^n)$. In this paper, we proposed techniques to improve both the time complexity and practical performance for maximum k-defective clique computation.

The problem of enumerating all maximal k-defective cliques was also studied recently where the Pivot+ algorithm proposed in [14] runs in $O^*(\gamma_k^n)$ time, the same as the time complexity of kDC. However, we remark that (1) Pivot+ is inefficient for finding the maximum k-defective clique in practice due to lack of pruning techniques; (2) Pivot+ achieves the time complexity via using a different branching technique from kDC and it is unclear how to improve the base of its time complexity without changing the branching rule. The problem of approximately counting k-defective cliques of a particular size, for the special case of $k = 1$ and 2, was studied in [22]; however, the techniques of [22] cannot be used for finding the maximum k-defective clique and for a general k.

9 CONCLUSION

In this paper, we proved an improved time complexity for the existing algorithm kDC [8]. We showed that for graphs with $\omega_k(G) \geq k+2$, a maximum k-defective clique can be found in $O^*((\alpha\Delta)^{k+2} \times \gamma_{k-1}^{\alpha})$ time and also in $O^*((\alpha\Delta)^{k+2} \times (k+1)^{\alpha+k+1-\omega_k(G)})$ time. In addition, we designed a new degree-sequence-based reduction rule that can be conducted in linear time, and theoretically demonstrated its effectiveness compared with other reduction rules. The practical efficiency of our algorithm and techniques are demonstrated on three benchmark collections with 290 graphs in total.

ACKNOWLEDGMENTS

The author is supported by the Australian Research Council Fundings of 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] 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.

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

[5] 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.

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

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

[8] Lijun Chang. 2023. Efficient Maximum K-Defective Clique Computation with Improved Time Complexity. *Proc. ACM Manag. Data* 1, 3, Article 209 (nov 2023), 26 pages. https://doi.org/10.1145/3617313

[9] Lijun Chang, Rashmika Gamage, and Jeffrey Xu Yu. 2024. Efficient k-Clique Count Estimation with Accuracy Guarantee. *Proc. VLDB Endow.* 17, 11 (2024), 3707–3719.

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

[11] Lijun Chang, Jeffrey Xu Yu, and Lu Qin. 2012. Fast Maximal Cliques Enumeration in Sparse Graphs. *Algorithmica* (2012).

[12] 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.

[13] James Cheng, Yiping Ke, Ada Wai-Chee Fu, Jeffrey Xu Yu, and Linhong Zhu. 2011. Finding maximal cliques in massive networks. *ACM Trans. Database Syst.* 36, 4 (2011), 21:1–21:34.

[14] Qiangqiang Dai, Rong-Hua Li, Meihao Liao, and Guoren Wang. 2023. Maximal Defective Clique Enumeration. *Proc. ACM Manag. Data* 1, 1 (2023), 77:1–77:26. https://doi.org/10.1145/3588931

[15] 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).

[16] Uriel Feige, Guy Kortsarz, and David Peleg. 2001. The Dense k-Subgraph Problem. *Algorithmica* 29, 3 (2001), 410–421. https://doi.org/10.1007/S004530010050

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

[18] 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.

[19] 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.

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

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

[22] 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.

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

[24] Mingming Jin, Jiongzhi Zheng, and Kun He. 2024. KD-Club: An Efficient Exact Algorithm with New Coloring-Based Upper Bound for the Maximum k-Defective Clique Problem. In *Proc. of AAAI'24*. 20735–20742.

[25] 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.

[26] 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*.

[27] 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.

[28] 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.

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

[30] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. 2012. *Foundations of Machine Learning*. MIT Press.

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

[32] 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.

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

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

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

[36] 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).

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

[38] 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.

[39] 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.

[40] 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.

[41] 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.

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

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

[44] 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.

[45] 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.

[46] Niklas Wünsche. 2021. *Mind the Gap When Searching for Relaxed Cliques*. Ph.D. Dissertation. Philipps-Universität Marburg.

[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.