Maximum k-Plex Computation: Theory and Practice

LIJUN CHANG, The University of Sydney, Australia
KAI YAO, The University of Sydney, Australia

The k-plex model relaxes the clique model by allowing each vertex to miss up to k neighbors, including the vertex itself. A 1-plex is a clique. Many exact algorithms have been recently designed for finding the k-plex with the largest number of vertices, known as the maximum k-plex computation problem. However, all the existing algorithms, except BS, has the trivial worst-case time complexity of $O^*(2^n)$ when ignoring polynomial factors. On the other hand, although BS improves the time complexity to $O^*(\beta_k^n)$ where $\beta_k < 2$ is a constant depending only on k, its practical performance is not satisfactory. In this paper, we study the maximum k-plex computation problem from both theory and practice. We first propose two new reduction rules and a new branching rule and prove that the base of the exponential time complexity is reduced to γ_k when the new reduction and branching rules are incorporated into a standard backtracking algorithm; here $\gamma_k < \beta_k$. We then design a two-stage approach kPlexT to improve the exponent of the time complexity by separating the search of large k-plexes from the search of small ones. We prove that kPlexT runs in $O^*((\alpha\Delta)^{k+1}\gamma_k^\alpha)$ time when the maximum k-plex size $\omega_k(G)$ is at least $2k-1$, and in $O^*((\alpha\Delta)^{k+1}\gamma_k^\alpha + \min\{\gamma_k^n, n^{2k-2}\})$ time otherwise; here, α is the degeneracy and Δ is the maximum degree of the input graph G. We also prove that with slight modification, kPlexT runs in $O^*((\alpha\Delta)^{k+1}(k+1)^{\alpha+k-\omega_k(G)})$ time when $\omega_k(G) \geq 2k-1$. Finally, we propose another reduction rule and a better initialization method to improve the practical performance of kPlexT. Extensive empirical studies demonstrate that kPlexT achieves state-of-the-art practical performance. We also show that our improved time complexity carries over to other related problems such as enumerating all maximal k-plexes, quasi-cliques, and k-biplexes.

CCS Concepts: • **Information systems → Social networks**; • **Theory of computation → Graph algorithms analysis**.

Additional Key Words and Phrases: k-plex, exact exponential time algorithm

ACM Reference Format:
Lijun Chang and Kai Yao. 2024. Maximum k-Plex Computation: Theory and Practice. *Proc. ACM Manag. Data* 2, 1 (SIGMOD), Article 63 (February 2024), 26 pages. https://doi.org/10.1145/3639318

1 INTRODUCTION

Finding dense (i.e., cohesive) subgraphs in a large graph is a fundamental problem in graph mining with diverse applications, such as community detection in social networks [5], real-time story identification in social media [2], anomaly detection in financial networks [1] and protein complexes detection in biological networks [30]. Different models have been formulated in the literature for measuring the cohesiveness of a graph, such as clique [8], average degree [32], k-core [4], k-truss [33], and so on. Among them, the clique model, requiring all vertices to be connected to each other (i.e., a clique is a complete graph), is naturally the most cohesive subgraph model. A large body of literature has been dedicated to clique-related problems, such as maximal clique enumeration [8, 15, 31] and maximum clique computation [9, 22, 23, 28].

Authors' addresses: Lijun Chang, Lijun.Chang@sydney.edu.au, The University of Sydney, Sydney, Australia; Kai Yao, Kai.Yao@sydney.edu.au, The University of Sydney, Sydney, Australia.

ACM 2836-6573/2024/2-ART63
https://doi.org/10.1145/3639318

The clique model however may be overly restrictive for many application scenarios, by noting that noise and faults naturally exist in the data acquisition process. Consequently, large and cohesive subgraphs can hardly be detected as cliques in some applications. In view of this, the k-plex model has been formulated in the literature [29], which relaxes the clique concept by allowing each vertex to miss up to k neighbors, including the vertex itself. A 1-plex is simply a clique. Same as the clique-related problems, both the maximal k-plex enumeration problem and the maximum k-plex computation problem have been extensively studied recently, e.g., see [3, 7, 10, 11, 13, 17, 19, 24, 26, 29, 34, 36–38, 41].

The maximum k-plex computation problem aims to find a k-plex with the largest number of vertices. Despite being an NP-hard problem [21], several exact and practical algorithms have been proposed recently, e.g., BS [38], BnB [17], Maplex [41], KpLeX [19], and kPlexS [10]. Among them, KpLeX and kPlexS are the two most recent and also most efficient algorithms, able to find the maximum k-plex in many large real-world graphs. However, both algorithms have their deficiencies. kPlexS makes use of an important property of large k-plexes (i.e., all k-plexes of size at least $2k-1$ have diameters at most 2) to dramatically reduce the search space, and thus is designed to be only able to find maximum k-plexes of size at least $2k-1$. KpLeX, though being able to find the maximum k-plex regardless of its size, performs much worse than kPlexS when the maximum k-plex size is at least $2k-1$. Furthermore, none of the existing exact maximum k-plex computation algorithms, except BS [38], beat the trivial time complexity of $O^*(2^n)$ where n is the number of vertices in the input graph G. [1] BS runs in $O^*(\beta_k^n)$ time where β_k is a constant that depends only on k; example values are $\beta_2 = 1.839$, $\beta_3 = 1.928$, $\beta_4 = 1.966$, and $\beta_5 = 1.984$. But, BS's practical performance is not satisfactory and it runs significantly slower than BnB [17] and KpLeX [19] as demonstrated in [17, 19].

In this paper, we focus on the maximum k-plex computation problem and propose a new algorithm that not only improves the state-of-the-art time complexity for exact maximum k-plex computation but also runs faster than the existing algorithms in practice. We present our techniques gradually by first introducing the ones that are required to achieve the time complexity and then the ones for practical efficiency consideration. Firstly, we propose two new reduction rules and a new branching rule and prove that the base of the exponential time complexity is reduced to γ_k when the new reduction and branching rules are incorporated into a standard backtracking algorithm; here $\gamma_k < 2$ is a constant that depends only on k and is smaller than β_k, e.g., $\gamma_2 = 1.7549$, $\gamma_3 = 1.8886$, $\gamma_4 = 1.9479$, and $\gamma_5 = 1.9751$. We remark that while the algorithm itself retains its simplicity, the intricacy lies within the time complexity analysis. Secondly, we design a two-stage approach kPlexT to improve the exponent of the time complexity by separating the search of large k-plexes from the search of small ones. Specifically, kPlexT first in Stage-I tries to find a maximum k-plex that is of size at least $2k-1$, and then goes into Stage-II to find a maximum k-plex of size at most $2k-2$ if Stage-I fails. Note that, Stage-I is expected to run faster than Stage-II, since Stage-I can utilize the diameter-two property of large k-plexes as kPlexS does. As a result, kPlexT is able to find the maximum k-plex regardless of its size and runs faster when the maximum k-plex size is at least $2k-1$. We prove that kPlexT runs in $O^*((\alpha\Delta)^{k+1}\gamma_k^\alpha)$ time when the maximum k-plex size, denoted $\omega_k(G)$, is at least $2k-1$, and runs in $O^*((\alpha\Delta)^{k+1}\gamma_k^\alpha + \min\{\gamma_k^n, n^{2k-2}\})$ time otherwise; here, Δ is the maximum degree of the input graph G, and α is the degeneracy of G which is at most $O(\sqrt{m})$ and is small in practice [15] where m is the number of edges in G. We also prove that with slight modification, kPlexT runs in $O^*((\alpha\Delta)^{k+1}(k+1)^{\alpha+k-\omega_k(G)})$ time when $\omega_k(G) \geq 2k-1$, matching the time complexity of kPlex-gap [35] that is proposed concurrent to this paper. Thirdly, to improve

[1] For presentation simplicity, we also use the O^* notation, in addition to the usual O notation, for time complexity analysis. The O^* notation focuses on the exponential part of the time complexity by hiding factors that are polynomial to the input.

Table 1. A comparison of the time complexity for maximum k-plex computation and maximal k-plex enumeration algorithms that have nontrivial time complexities (the O^* notation hides polynomial factors; β_k and γ_k are constants smaller than 2 that only depend on k; $\gamma_k < \beta_k$; α is the degeneracy and Δ is the maximum degree of G; $\omega_k(G)$ is the maximum k-plex size of G)

Algorithm	Time complexity	Problem	Limitation
BS [38]	$O^*(\beta_k^n)$	Maximum k-plex computation	None
FaPlexen [42]	$O^*(\beta_k^n)$	Maximal k-plex enumeration	None
ListPlex [36]	$O^*((\alpha\Delta)^{k+1}\beta_k^\alpha)$	Maximal k-plex enumeration	k-plex size $\geq 2k-1$
FP [13]	$O^*(\beta_k^{\alpha\Delta})$	Maximal k-plex enumeration	k-plex size $\geq 2k-1$
kPlex-gap [35]	$O^*((\alpha\Delta)^{k+1}(k+1)^{\alpha+k-\omega_k(G)})$	Maximum k-plex computation	k-plex size $\geq 2k-1$
kPlexT	$O^*((\alpha\Delta)^{k+1}\gamma_k^\alpha)$	Both problems	k-plex size $\geq 2k-1$
kPlexT	$O^*\big((\alpha\Delta)^{k+1}\gamma_k^\alpha + \min\{\gamma_k^n, n^{2k-2}\}\big)$	Both problems	None
kPlexT	$O^*((\alpha\Delta)^{k+1}(k+1)^{\alpha+k-\omega_k(G)})$	Maximum k-plex computation	k-plex size $\geq 2k-1$

the practical performance of kPlexT, we propose another reduction rule to remove unpromising candidate vertices and a new initialization method to find a large k-plex initially.

Contributions. Our main contributions are summarized as follows.

- We propose two new reduction rules and a new branching rule, and prove that a standard backtracking algorithm after incorporating our new reduction and branching rules runs in $O^*(\gamma_k^n)$ time where $\gamma_k < \beta_k$. This improves the base of the exponential time complexity.
- We design a two-stage approach kPlexT, and prove that it runs in $O^*((\alpha\Delta)^{k+1}\gamma_k^\alpha)$ time when the maximum k-plex size $\omega_k(G)$ is at least $2k-1$ and in $O^*((\alpha\Delta)^{k+1}\gamma_k^\alpha + \min\{\gamma_k^n, n^{2k-2}\})$ time otherwise. This improves the exponent of the time complexity. We also prove that with slight modification, kPlexT runs in $O^*((\alpha\Delta)^{k+1}(k+1)^{\alpha+k-\omega_k(G)})$ time, by using the degeneracy gap parameterization, when $\omega_k(G) \geq 2k-1$.
- To make kPlexT practically efficient, we propose another reduction rule to remove unpromising candidate vertices and a new initialization method to find a large k-plex initially.
- We conduct extensive experimental studies on two graph collections consisting of 223 graphs, which demonstrate that kPlexT achieves state-of-the-art practical performance. For example, with a time limit of 30 minutes and for $k = 10$, kPlexT solves 58 graphs from the DIMACS10 collection, while the existing algorithms can only solve up to 36 graphs.
- We show that our improved time complexity carries over to other related problems such as enumerating all maximal k-plexes, quasi-cliques and k-biplexes.

A summary of the various time complexities of kPlexT in comparison to the state-of-the-art maximum k-plex computation algorithms and maximal k-plex enumeration algorithms is given in Table 1.

Organization. The remainder of the paper is organized as follows. Related works are briefly reviewed in below. We present preliminaries of the studied problem in Section 2. Our algorithms are presented in Section 3, and experimental results are reported in Section 4. We discuss how to extend our time complexity to other related problems in Section 5. Finally, Section 6 concludes the paper.

Related Works. As a relaxation of clique, the k-plex model was formulated in [29]. Although detecting the maximum k-plex is an NP-hard problem [21], several exact algorithms have been proposed and implemented in the literature. Balasundaram et al. [3] proposed a branch-and-cut algorithm IPBC by exploiting graph-theoretical properties of k-plex. McClosky and Hicks [24]

proposed the OsterPlex algorithm by borrowing ideas from the maximum clique computation literature. Moser et al. [26] developed the GuidedBranching algorithms based on methods from parameterized algorithms. Xiao et al. [38] proposed the BS algorithm that runs in $O^*(\beta_k^n)$ time. Gao et al. [17] proposed the BnB algorithm by designing novel reduction, branching and upper bounding techniques. Zhou et al. [41] proposed the Maplex algorithm by utilizing second-order reduction and graph color bounding techniques. Jiang et al. [19] designed the KpLeX algorithm by proposing a new upper bound technique. Chang et al. [10] developed the kPlexS algorithm by proposing a more efficient preprocessing algorithm as well as incorporating second-order techniques in the branch-and-bound search. As discussed above, none of the existing algorithms, except BS, beat the trivial time complexity of $O^*(2^n)$, while BS does not perform well in practice. In this paper, we bridge theory and practice by proposing the kPlexT algorithm that simultaneously achieves state-of-the-art time complexity and practical performance. Concurrent to our work, Wang et al. [35] proposed the kPlex-gap algorithm and analyzed its time complexity by using the degeneracy gap (i.e., the difference between the degeneracy-based upper bound $\alpha + k$ and the maximum k-plex size $\omega_k(G)$) parameterization. Specifically, kPlex-gap runs in $O^*((\alpha\Delta)^{k+1}(k+1)^{\alpha+k-\omega_k(G)})$ time. However, kPlex-gap can only handle graphs with $\omega_k(G) \geq 2k-1$, and the time complexity is worse than $O^*((\alpha\Delta)^{k+1}2^{\alpha})$ when $(\alpha + k - \omega_k(G))\log_2(k+1) > \alpha$ which is the case for many of our tested graphs. Nevertheless, we will prove that the time complexity of our algorithm kPlexT is also bounded by $O^*((\alpha\Delta)^{k+1}(k+1)^{\alpha+k-\omega_k(G)})$ when $\omega_k(G) \geq 2k-1$.

A highly related problem is the maximal k-plex enumeration problem, which aims to enumerate all maximal k-plexes in a graph and has also been extensively studied [11–13, 34, 36, 37, 42]. For the enumeration problem, most of the existing algorithms followed the backtracking paradigm of the Bron-Kerbosch algorithm [8], a classic algorithm for maximal clique enumeration, and run in $O^*(2^n)$ time. FaPlexen proposed in [42] is the first maximal k-plex enumeration algorithm that beats the $O^*(2^n)$ time complexity. Specifically, FaPlexen runs in $O^*(\beta_k^n)$ time. As enumerating all maximal k-plexes is intrinsically more time-consuming than computing the maximum k-plex, recent efforts have been spent on enumerating only large maximal k-plexes, i.e., of size at least τ for a user-given parameter τ that is at least $2k-1$. For the problem of enumerating all *large* maximal k-plexes, Wang et al. [36] developed the ListPlex algorithm with a time complexity of $O^*((\alpha\Delta)^{k+1}\beta_k^{\alpha})$, and Dai et al. [13] proposed the FP algorithm with a time complexity of $O^*(\beta_k^{\alpha\Delta})$. As the largest one among all maximal k-plexes is a maximum k-plex, these enumeration algorithms can be used for computing the maximum k-plex. However, as demonstrated in our experimental studies in Section 4, even the most recent algorithm FP that incorporates pruning techniques based on the size lower bound τ does not scale to large k. We remark that our algorithm kPlexT also achieves a better time complexity for the maximal k-plexes enumeration problem, as summarized in Table 1.

The k-plex model is related to the λ-quasi-clique model, which allows each vertex in a λ-quasi-clique g to miss up to $\lfloor(1-\lambda)|V(g)| + \lambda\rfloor$ neighbors, including the vertex itself. That is, a λ-quasi-clique g is a k-plex where the value of k is a function of $|V(g)|$ and λ. However, quasi-clique-related problems are generally harder, since a subgraph of a λ-quasi-clique may be not a λ-quasi-clique. Branch-and-bound algorithms have been designed in [18, 40] for enumerating all large maximal λ-quasi-cliques. Recently, the k-plex model was also extended to the k-biplex model for bipartite graphs, which allows each vertex to have up to k non-neighbors in the other side of the biplex. Exact algorithms have been designed for enumerating all maximal k-biplexes in [14] and for computing maximum k-biplex in [39]. As we will show in Section 5, our techniques can be used to achieve better time complexities for enumerating all maximal quasi-cliques and all maximal k-biplexes.

2 PRELIMINARIES

In this paper, we consider an unweighted and undirected graph $G = (V, E)$ without self-loops and parallel edges, where V represents the set of vertices and E represents the set of undirected edges in G. We denote the number of vertices and the number of undirected edges in G by n and m, respectively, i.e., $n = |V|$ and $m = |E|$. For an undirected edge (u, v) between u and v, we say that u (resp. v) is adjacent to and a neighbor of v (resp. u). For a vertex $u \in V$, its set of neighbors in G is denoted by $N_G(u) = \{v \in V \mid (u, v) \in E\}$, and its *degree* in G is $d_G(u) = |N_G(u)|$. The set of u's non-neighbors in G is denoted $\overline{N}_G(u) = V \setminus (N_G(u) \cup \{u\})$. Note that ***a vertex is neither a neighbor nor a non-neighbor of itself***. Given a vertex subset $S \subseteq V$, the subgraph of G induced by S is denoted by $G[S] = (S, \{(u, v) \in E \mid u, v \in S\})$. For ease of presentation, we omit the subscript G from the notations when the context is clear. For an arbitrary graph g, we use $V(g)$ and $E(g)$ to represent its set of vertices and its set of edges, respectively.

Definition 2.1 (k-plex). A graph g is a k-plex if every vertex $u \in V(g)$ has at least $|V(g)| - k$ neighbors in g, i.e., $|N_g(u)| \geq |V(g)| - k$. Equivalently, g is a k-plex if every vertex $u \in V(g)$ has at most $k - 1$ non-neighbors in g, i.e., $|\overline{N}_g(u)| \leq k - 1$.

It is easy to see that for any k-plex g in G, the subgraph of G induced by $V(g)$ is also a k-plex. Thus, we ***refer to a k-plex by its set of vertices***, and measure the size of a k-plex by its number of vertices. The property of k-plex is hereditary; that is, any (vertex) subset of a k-plex is also a k-plex. A k-plex $P \subseteq V$ is *maximal* if none of its proper superset is a k-plex, and P is a *maximum k-plex* in G if its size is the largest among all k-plexes of G. Note that a maximum k-plex is not unique. Consider the graph in Figure 1, $\{v_1, v_2, v_3, v_4\}$ and $\{v_6, v_7, v_8, v_9\}$ are two maximum 2-plexes of equal size.

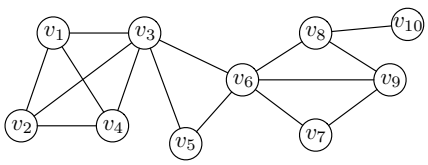


Fig. 1. An example graph

Problem Statement. Given a graph $G = (V, E)$ and an integer $k \geq 2$, we study the problem of maximum k-plex computation, aiming to find the largest k-plex in G.

Frequently used notations are summarized in Table 2.

Table 2. Frequently used notations

Notation	Meaning
$P, S \subseteq V$	k-plexes
$N_S(u)$	the subset of u's neighbors that are in S, i.e., $N_S(u) = N_G(u) \cap S$
$\overline{N}_S(u)$	the subset of u's non-neighbors that are in S, i.e., $\overline{N}_S(u) = S \setminus (N_S(u) \cup \{u\})$
$d_S(u)$	the cardinality of $N_S(u)$, i.e., $d_S(u) = \lvert N_S(u)\rvert$

3 OUR APPROACHES

In this section, we propose efficient algorithms with improved time complexity for exact maximum k-plex computation. As the problem is NP-hard [21], our algorithms run in exponential time in

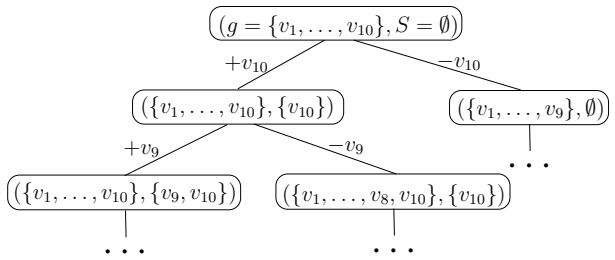


Fig. 2. Running example of a backtracking algorithm

the worst case. In the following, we first in Section 3.1 propose techniques to improve the base of the exponential time complexity, and then in Section 3.2 propose a two-stage approach kPlexT to improve the exponent of the time complexity. Lastly, we propose pruning techniques in Section 3.3 to make our algorithm kPlexT practical.

3.1 Improving the Base of the Exponential Time Complexity

In this subsection, we propose a new *backtracking* algorithm, also called *branch-and-bound* search algorithm, kPlexBB to improve the base of the exponential time complexity for exact maximum k-plex computation. The general idea of a backtracking algorithm is iteratively growing a partial solution S by including or discarding vertices. Specifically, an instance of the backtracking is denoted (g, S) where $S \subseteq V(g)$ is a k-plex in g, and solving the instance (g, S) means finding the largest k-plex in the instance (i.e., in g and contains S). Thus, solving the instance $(G, \emptyset)$ finds the largest k-plex in G. To solve an instance (g, S), a ***branching vertex*** u is selected from $V(g) \setminus S$, referred to as the set of ***candidate vertices***, and then two new instances are generated: $(g, S \cup u)$ that includes u into the solution, and $(g \setminus u, S)$ that discards u from the graph (and thus also from the solution). Here $S \cup u$ denotes the union of S and u, and $g \setminus u$ denotes the resulting graph after removing u and its associated edges from g. Solving $(g, S \cup u)$ and $(g \setminus u, S)$ then solves (g, S). A running example of computing the maximum k-plex in the graph in Figure 1 for $k = 2$ by a standard backtracking algorithm is shown in Figure 2, where the graph g in an instance (g, S) is represented by its vertex set. For the instance $(g = \{v_1, \ldots, v_{10}\}, S = \emptyset)$, assume the branching vertex is selected as v_{10}, it then generates the instance $(g = \{v_1, \ldots, v_{10}\}, S = \{v_{10}\})$ that includes the branching vertex v_{10} into the solution and the instance $(g = \{v_1, \ldots, v_9\}, S = \emptyset)$ that removes v_{10} from the graph. Similarly, two new instances are generated based on the branching vertex v_9 for the first instance.

Different backtracking algorithms usually differ in their strategies of selecting the branching vertex; this is referred to as ***branching rules***. In addition, they may also use different ***reduction rules*** and ***upper bound-based pruning***. Specifically, applying reduction rules to an instance (g, S) will generate a new instance (g', S') such that (i) $V(g') \setminus S' \subseteq V(g) \setminus S$ and (ii) an optimal solution of (g, S) can be obtained from an optimal solution of (g', S'). That is, some of the candidate vertices $V(g) \setminus S$ are added to S and some are removed from g. For example, for the bottom-left instance $(g = \{v_1, \ldots, v_{10}\}, S = \{v_9, v_{10}\})$ in Figure 2 and $k = 2$, all vertices of $\{v_1, \ldots, v_7\}$ can be removed from g since each of them does not form a valid k-plex with S. Consequently, applying reduction rules potentially reduces the number of candidate vertices and thus the search space. On the other hand, upper bound-based pruning computes an upper bound of the largest k-plex in an instance, and prunes the instance if the computed upper bound is no larger than the currently found largest k-plex.

The better time complexity of our algorithm kPlexBB is achieved by our new branching rule that we are going to propose shortly, and the following three reduction rules.

Lemma 3.1 (Reduction Rule **RR1**). *Given an instance (g, S) and a vertex $u \in V(g) \setminus S$, if $S \cup u$ is not a k-plex, then none of the maximal k-plexes in the instance contains u and thus we can remove u from g.*

RR1 is trivial and has been used in the existing algorithms.

Lemma 3.2 (Reduction rule **RR2**). *Given an instance (g, S) and a vertex $u \in V(g) \setminus S$, if u as well as all of its non-neighbors in g have at most $k-1$ non-neighbors in g, then every maximal k-plex in the instance must contain u and thus we can greedily add u to S.*

Proof. We prove this lemma by contradiction. Suppose the instance has a maximal k-plex P that does not contain u. Firstly, for any vertex $v \in P$ that is a neighbor of u, we have $|\overline{N}_{P\cup u}(v)| = |\overline{N}_P(v)| \leq k-1$ since P is a k-plex. Secondly, for any vertex $v \in P \cup u$ that is not a neighbor of u, we have $|\overline{N}_{P\cup u}(v)| \leq |\overline{N}_g(v)| \leq k-1$, where the second inequality follows from the assumption of the lemma. Thus, $P \cup u$ is also a k-plex, contradicting that P is a maximal k-plex. Hence, the lemma holds. □

A special case of **RR2** is that if u has no non-neighbors in g, then we can greedily add u to S.

Lemma 3.3 (Reduction Rule **RR3**). *Given an instance (g, S) and a vertex $u \in V(g) \setminus S$, if u has exactly one non-neighbor in g and its non-neighbor v has exactly k non-neighbors in g, then any maximal k-plex that is in the instance and does not contain u must include v and all of v's non-neighbors (excluding u).*

Proof. We prove the lemma by contradiction. Suppose the instance has a maximal k-plex P that does not contain u and also some vertex of $(\overline{N}_g(v) \cup v) \setminus u$. Firstly, if $v \notin P$, then u is adjacent to all vertices of P and thus $P \cup u$ is a k-plex, contradicting that P is maximal. Secondly, if $v \in P$ and $|P \cap \overline{N}_g(v)| \leq |\overline{N}_g(v)| - 2 = k-2$, then $|\overline{N}_{P\cup u}(v)| \leq k-1$ and thus $P \cup u$ is also a k-plex, contradicting that P is maximal. Hence, the lemma holds. □

The pseudocode of our algorithm kPlexBB is shown in Algorithm 1, which invokes the recursive procedure Branch&Bound with an initially empty solution P. When the algorithm finishes, P stores the maximum k-plex. In Branch&Bound, we first apply reduction rules (e.g., **RR1**–**RR3**) to reduce the instance (g, S) to (g', S') at Line 4. We then check whether g' itself is a k-plex. If it is, we update P by $V(g')$ (Lines 5–6). Otherwise, we choose a branching vertex b by invoking the procedure ChooseBranchingVertex (Line 8), and generate two branches/instances based on b: one branch adds b into the solution S' (Line 9), and the other branch removes b from g' (Line 10). The procedure ChooseBranchingVertex implementing our branching rule is shown in Lines 11–19 of Algorithm 1. Firstly, we get the set D of all vertices that have at least k non-neighbors in g (Line 11). Then, from D we only keep the ones that have the fewest neighbors in S and denote the resulting set by D' (Line 12). That is, let $\delta = \min_{v\in D} d_S(v)$ be the minimum value among the degrees, computed in S, of all vertices of D, then $D' = \{v \in D \mid d_S(v) = \delta\}$. Let u be the vertex of D' that has the smallest degree in g (Line 13), i.e., $u = \arg\min_{v\in D'} d_g(v)$. The branching vertex b is selected based on u by considering three different cases.

- If $u \in S$, then b is any one of u's non-neighbors in $V(g) \setminus S$ (Line 14).
- Otherwise, $u \notin S$.
  - If $d_S(u) < |S|$ or $|\overline{N}_g(u)| > k$, then b is set as u (Line 15).
  - Otherwise, $d_S(u) = |S|$ and $|\overline{N}_g(u)| = k$. Let w be the vertex of $V(g) \setminus S$ that has the largest degree in g (Line 17).

Algorithm 1: kPlexBB(G, k)

Input: A graph G and an integer $k \geq 2$
Output: A maximum k-plex in G

1 $P \leftarrow \emptyset$;
2 Branch&Bound$(G, k, \emptyset, P)$;
3 **return** P;

Procedure Branch&Bound(g, k, S, P)
4 $(g', S') \leftarrow$ apply reduction rules to (g, S); /* e.g., **RR1–RR3** */;
5 **if** g' *is a* k*-plex* **then**
6 **if** $|V(g')| > |P|$ **then** $P \leftarrow V(g')$;
7 **else**
8 $b \leftarrow$ ChooseBranchingVertex(g', k, S');
9 Branch&Bound$(g', k, S' \cup \{b\}, P)$; /* Add b into S' */;
10 Branch&Bound$(g' \setminus \{b\}, k, S', P)$; /* Remove b from g' */;

Procedure ChooseBranchingVertex(g, k, S)
11 $D \leftarrow$ the set of g's vertices that have at least k non-neighbors in g;
12 $D' \leftarrow$ the subset of D whose degree in S is minimum;
13 $u \leftarrow$ a vertex of D' that has the smallest degree in g;
14 **if** $u \in S$ **then return** any one of u's non-neighbors in $V(g) \setminus S$;
15 **else if** $d_S(u) < |S|$ **or** $|\overline{N}_g(u)| > k$ **then return** u;
16 **else**
17 $w \leftarrow$ a vertex of $V(g) \setminus S$ that has the largest degree in g;
18 **if** $|\overline{N}_g(w)| = 1$ **then return** w;
19 **return** any one of w's non-neighbors that are in $V(g) \setminus S$ and has at least k non-neighbors in g;

- If $|\overline{N}_g(w)| = 1$, then b is set as w (Line 18).
- Otherwise, b is any one of w's non-neighbors that are in $V(g) \setminus S$ and has at least k non-neighbors in g (Line 19). Such a vertex always exist since (1) w is the vertex with the largest degree in g and (2) we have exhaustively applied reduction rule **RR2** at Line 4.

Note that, if we do not apply reduction rules at Line 4 and let ChooseBranchingVertex simply return an arbitrary vertex, then Algorithm 1 is a standard backtracking algorithm.

Time Complexity Analysis of kPlexBB**.** To analyze the time complexity of kPlexBB, we will consider the (binary) search tree $\mathcal{T}$ of recursively running Branch&Bound, where a snippet is shown in Figure 3. To avoid confusion, we refer to nodes of the search tree $\mathcal{T}$ by *nodes*, and vertices of a graph by *vertices*. Each node of $\mathcal{T}$ corresponds to an instance of Branch&Bound, i.e., (g, S), and has two children: the left child includes the branching vertex b into the solution S (corresponding to Line 9 of Algorithm 1), and the right child excludes b from the graph g (corresponding to Line 10). Note that, each child may also include or exclude other vertices due to applying reduction rules at Lines 4–5, e.g., **RR1**–**RR3**; for presentation simplicity, we omit these details from Figure 3. Note that, we also consider Line 5 as a reduction rule: if g is a k-plex, then we add all vertices of $V(g) \setminus S$ to S. 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 instance to which I corresponds; we abbreviate $I_i.g$ and $I_i.S$ as g_i and S_i, respectively. *Note that $I.g$ and $I.S$ denote the ones obtained after applying the reduction rules at Line 4 of Algorithm 1, not the ones input to* Branch&Bound. We measure the

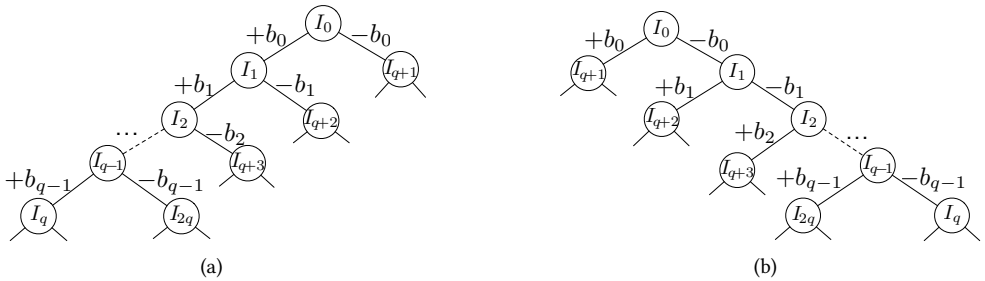


Fig. 3. Snippets of the (binary) search tree

size of I by its number of candidate vertices, denoted $|I| = |V(I.g)| - |I.S|$. Note that, $|I| = 0$ for leaf nodes and for any child I' of I, it holds that $|I'| \leq |I| - 1$.

Lemma 3.4. *Let $\mathcal{T}$ be the search tree of running Algorithm 1. 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 $x^{k+3} - 2x^{k+2} + x^2 - x + 1 = 0$. For example $\gamma_2 = 1.7549$, $\gamma_3 = 1.8886$, $\gamma_4 = 1.9479$, and $\gamma_5 = 1.9751$.*

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 $I_0 = (g_0, S_0)$, let u_0 be the vertex that is chosen at Line 13 for the instance. Assuming that $\ell(I') \leq \gamma_k^{|I'|}$ holds for every proper descendant I' of I_0, we prove $\ell(I_0) \leq \gamma_k^{|I_0|}$ by considering three cases, corresponding to Lines 14–16.
Case-I: $u_0 \in S_0$ (the case of Line 14) or $d_{S_0}(u_0) < |S_0|$ (the first sub-case of Line 15). Let's consider the path $(I_0, I_1, \ldots, I_q)$ that starts from I_0, always visits left child and stops once reaching a node $I_q \neq I_0$ satisfying $|I_q| \leq |I_{q-1}| - 2$; note that, such an I_q always exists by noting that g_0 is not a k-plex. It is trivial that $\ell(I_0) = \ell(I_{q+1}) + \ell(I_{q+2}) + \cdots + \ell(I_{2q}) + \ell(I_q)$; see Figure 3(a) for the nodes $\{I_{q+1}, \ldots, I_{2q}\}$. Recall that $I_i = (g_i, S_i)$. The definition of the path $(I_0, I_1, \ldots, I_q)$ implies that for each $1 \leq i \leq q - 1$,

- $|I_i| = |I_{i-1}| - 1 = |I_0| - i$.
- $S_i = S_{i-1} \cup v$, for some vertex $v \in V(g_{i-1}) \setminus S_{i-1}$.
- $g_i = g_0$.

Thus, the following two facts hold.

Fact 1.1. For $q + 1 \leq i \leq 2q$, $|I_i| \leq |I_{i-q-1}| - 1 = |I_0| + q - i$.
Fact 1.2. $|I_q| \leq |I_{q-1}| - 2 = |I_0| - q - 1$.

For $0 \leq i \leq q - 1$, let δ_i be the minimum value among the degrees, computed in S_i, of all vertices of g_i that have at least k non-neighbors in g_i; that is, $\delta_i = \min_{v \in V(g_i): |\overline{N}_{g_i}(v)| \geq k} d_{S_i}(v)$, and $\delta_i = d_{S_i}(u_i)$ where u_i is the vertex chosen at Line 13 for (g_i, S_i). It can be proved by induction that for $1 \leq i \leq q-1$, it holds that $\delta_i = \delta_0$ and moreover there is a vertex $v_i \in S_i$ satisfying $|\overline{N}_{g_i}(v_i)| \geq k$ and $d_{S_i}(v_i) = \delta_i = \delta_0$; consequently, $u_i \in S_i$ or $d_{S_i}(u_i) = \delta_0 \leq |S_0| < |S_i|$. Specifically, if $u_{i-1} \in S_{i-1}$, then the branching vertex selected for I_{i-1} is a non-neighbor of u_{i-1} and thus $\delta_i = \delta_{i-1}$, otherwise, $u_{i-1} \notin S_{i-1}$ and $S_i = S_{i-1} \cup u_{i-1}$ and $\delta_i = \delta_{i-1}$; in either case, u_{i-1} is such a v_i. As $|I_{q-1}| = |I_{q-2}| - 1$, the reduction rule **RR1** has no effect on I_{q-1} and thus for the vertex $v_i \in S_i$ satisfying $|\overline{N}_{g_i}(v_i)| \geq k$

where $i = q - 1$, we have

$$\begin{aligned} & |\overline{N}_{S_{q-1}}(v_{q-1})| \leq k-2 \\ \Longleftrightarrow & |S_{q-1}| - \delta_0 - 1 \leq k-2 \\ \Longleftrightarrow & |S_0| + q - 1 - \delta_0 - 1 \leq k-2 \\ \Longleftrightarrow & q \leq k - (|S_0| - \delta_0) = k - (|S_0| - d_{S_0}(u_0)) \end{aligned} \tag{1}$$

Since either $u_0 \in S_0$ or $d_{S_0}(u_0) < |S_0|$, we have

Fact 1.3. $q \leq k - 1$.

Based on Facts 1.1, 1.2 and 1.3, we have

$$\begin{aligned} \ell(I_0) &= \ell(I_{q+1}) + \ell(I_{q+2}) + \cdots + \ell(I_{2q}) + \ell(I_q) \\ &\leq \gamma_k^{|I_{q+1}|} + \gamma_k^{|I_{q+2}|} + \cdots + \gamma_k^{|I_{2q}|} + \gamma_k^{|I_q|} \\ &\leq \gamma_k^{|I_0|-1} + \gamma_k^{|I_0|-2} + \cdots + \gamma_k^{|I_0|-q-1} \\ &\leq \gamma_k^{|I_0|-1} + \gamma_k^{|I_0|-2} + \cdots + \gamma_k^{|I_0|-k} \end{aligned} \tag{2}$$

where the last term is $\leq \gamma_k^{|I_0|}$ if γ_k is no smaller than the largest real root of the equation $x^{|I_0|} - x^{|I_0|-1} - \cdots - x^{|I_0|-k} = 0$ [16].

Case-II: $u_0 \notin S_0$, $d_{S_0}(u_0) = |S_0|$ and $|\overline{N}_{g_0}(u_0)| \geq k+1$ (the second sub-case of Line 15). Let $(I_0, I_1, \ldots, I_q)$ be defined in the same way as in Case-I. Then, Facts 1.1 and 1.2 still hold, i.e.,

- For $q+1 \leq i \leq 2q$, $|I_i| \leq |I_{i-q-1}| - 1 = |I_0| + q - i$.
- $|I_q| \leq |I_{q-1}| - 2 = |I_0| - q - 1$.

In addition, Equation (1) also still holds, by the same argument as above. But now, we have $d_{S_0}(u_0) = |S_0|$, and thus we can only conclude $q \leq k$. If $q \leq k-1$, then Equation (2) still holds. We consider the scenario of $q = k$ and prove that $|I_q| \leq |I_{q-1}| - 3$ in the following. For $0 \leq i \leq q-1$, recall from the analysis of Case-I that $\delta_i = \delta_0 = |S_0|$. Similarly, we let σ_i be the minimum value among the degrees, *computed in* g_i, of all vertices of g_i that have at least k non-neighbors in g_i and exactly δ_i neighbors in S_i, i.e., $\sigma_i = \min_{v \in V(g_i): |\overline{N}_{g_i}(v)| \geq k \wedge d_{S_i}(v) = \delta_i} d_{g_i}(v)$, and $\sigma_i = d_{g_i}(u_i)$; we can also prove that $\sigma_i = \sigma_0 = |V(g_0)| - 1 - |\overline{N}_{g_0}(u_0)| \leq |V(g_0)| - k - 2$. Thus, for u_{q-1}, it holds that $d_{S_{q-1}}(u_{q-1}) = \delta_{q-1} = |S_0|$ and $d_{g_{q-1}}(u_{q-1}) = \sigma_{q-1} \leq |V(g_0)| - k - 2$, i.e., $|\overline{N}_{S_{q-1} \cup b_{q-1}}(u_{q-1})| = q - 1 = k - 1$ and $|\overline{N}_{g_{q-1}}(u_{q-1})| = |\overline{N}_{g_0}(u_0)| \geq k+1$ where b_{q-1} is the branching vertex selected for I_{q-1}. Moreover, $u_{q-1} \in S_q$ and $S_{q-1} \cup b_{q-1} \subseteq S_q$. Thus, according to reduction rule **RR1**, we have $|V(g_q)| \leq |V(g_{q-1})| - 2$ and $|I_q| \leq |I_{q-1}| - 3$. Hence, we have

$$\begin{aligned} \ell(I_0) &= \ell(I_{q+1}) + \ell(I_{q+2}) + \cdots + \ell(I_{2q}) + \ell(I_q) \\ &\leq \gamma_k^{|I_{q+1}|} + \gamma_k^{|I_{q+2}|} + \cdots + \gamma_k^{|I_{2q}|} + \gamma_k^{|I_q|} \\ &\leq \gamma_k^{|I_0|-1} + \gamma_k^{|I_0|-2} + \cdots + \gamma_k^{|I_0|-q} + \gamma_k^{|I_0|-q-2} \\ &\leq \gamma_k^{|I_0|-1} + \gamma_k^{|I_0|-2} + \cdots + \gamma_k^{|I_0|-k} + \gamma_k^{|I_0|-k-2} \end{aligned} \tag{3}$$

where the last term is $\leq \gamma_k^{|I_0|}$ if γ_k is no smaller than the largest real root of the equation $x^{|I_0|} - x^{|I_0|-1} - \cdots - x^{|I_0|-k} - x^{|I_0|-k-2} = 0$ [16].

Case-III: $u_0 \notin S_0$, $d_{S_0}(u_0) = |S_0|$ and $|\overline{N}_{g_0}(u_0)| = k$ (the case of Line 16). This means that (1) all vertices of S_0 have at most $k-1$ non-neighbors in g_0, (2) all vertices of $V(g_0) \setminus S_0$ that have at least k non-neighbors in g_0 are fully adjacent to S_0, and (3) all vertices of $V(g_0) \setminus S_0$ have at most k non-neighbors in g_0. Let's consider the path $(I_0, I_1, \ldots, I_q)$ that starts from I_0, always visits *right*

child and stops once reaching either a leaf node I_q or a node $I_q \neq I_0$ satisfying $|I_q| \leq |I_{q-1}| - 2$, see Figure 3(b). This implies that for each $1 \leq i \leq q-1$,

- $|I_i| = |I_{i-1}| - 1 = |I_0| - i$.
- $S_i = S_0$
- $g_i = g_{i-1} \setminus v$, for some vertex $v \in V(g_{i-1}) \setminus S_0$.

For $0 \leq i \leq q-1$, let η_i be the minimum number of non-neighbors, counted in g_i, among all vertices of $V(g_i) \setminus S_i$; note that, $1 \leq \eta_i \leq \eta_0 \leq k$ and the vertex w_i selected at Line 17 for I_i has exactly η_i non-neighbors. Let's consider I_i for $0 \leq i \leq q-2$. It must hold that $|\overline{N}_{g_i}(w_i)| > 1$ and w_i has at least one non-neighbor $v_i \in V(g_i) \setminus S_i$ satisfying $|\overline{N}_{g_i}(v_i)| = k$; this is due to our way of choosing w_i and the fact that the reduction rules **RR2** and **RR3** have no effect on I_{i+1}. Consequently, $g_{i+1} = g_i \setminus v_i$, $\eta_{i+1} = \eta_i - 1$, and $V(g_{q-1}) \setminus S_{q-1}$ must contain w_{q-2} and at least one of w_{q-2}'s non-neighbors (otherwise, **RR2** would add w_{q-2} to S_{q-1}). Thus, if I_q is a leaf node, we also have $|I_q| \leq |I_{q-1}| - 2$. Facts 1.1 and 1.2 still hold for Figure 3(b), i.e.,

- For $q+1 \leq i \leq 2q$, $|I_i| \leq |I_{i-q-1}| - 1 = |I_0| + q - i$.
- $|I_q| \leq |I_{q-1}| - 2 = |I_0| - q - 1$.

Following the above discussion, we have

$$1 \leq \eta_{q-1} = \eta_0 - q + 1 \leq k - q + 1 \tag{4}$$

and thus $q \leq k$. If $q \leq k-1$, then Equation (2) holds. We consider the scenario of $q = k$. Then, all inequalities in Equation (4) become equalities, i.e., $\eta_0 = k$ and $\eta_{q-1} = 1$. Thus, all vertices of $V(g_0) \setminus S_0$ are fully adjacent to all vertices of S_0 and have exactly k non-neighbors in g_0, and $|\overline{N}_{g_{q-1}}(w_{q-1})| = 1$ and $|\overline{N}_{g_{q-1}}(v_{q-1})| = k$ and $\overline{N}_{g_{q-1}}(v_{q-1}) \subset V(g_{q-1}) \setminus S_{q-1}$ where v_{q-1} is the unique non-neighbor of w_{q-1} in g_{q-1}. Consequently, $|I_q| \leq |I_{q-1}| - k - 1$ according to **RR3**. For $k \geq 2$, we reach the same inequality as Equation (3).

The largest real root of the equation $x^{|I_0|} - x^{|I_0|-1} - \cdots - x^{|I_0|-k} - x^{|I_0|-k-2} = 0$ is larger than that of $x^{|I_0|} - x^{|I_0|-1} - \cdots - x^{|I_0|-k} = 0$. Thus, the lemma holds. □

Lemma 3.5. *Let C be the time complexity of running* Branch&Bound *without going into the recursions, i.e., without Lines 9–10. The time complexity of Algorithm 1 is $O(C \times \gamma_k^n)$ and $O^*(\gamma_k^n)$.*

Proof. Since the search tree $\mathcal{T}$ of running Algorithm 1 is a full binary tree (i.e., each non-leaf node has exactly two children), the total number of nodes in $\mathcal{T}$ is at most twice the number of leaf nodes. Thus, the time complexity of Algorithm 1 is $\gamma_k^{|I_r|}$ times C, where C is the time complexity for each node of the search tree; here I_r is the root node of the search tree $\mathcal{T}$ and $|I_r| \leq n$. As C is a polynomial function of the input size, the lemma follows. □

It is easy to see that the procedure ChooseBranchingVertex (in Algorithm 1) for choosing the branching vertices runs in $O(|E(g)|)$ time assuming $|V(g)| \leq |E(g)|$. Exhaustively applying the reduction rules **RR1**–**RR3** can be conducted in $O(|E(g)|)$ time. Thus, the time complexity of kPlexBB is $O(m \times \gamma_k^n)$.

We can easily modify Algorithm 1 to determine whether G has a k-plex of size τ or not, i.e., terminate a Branch&Bound instance (g, k, S, P) once $|V(g)|$ reaches τ. We prove in the lemma below that the modified algorithm runs in $O(C \times (k+1)^{n-\tau})$ and $O^*((k+1)^{n-\tau})$ time, which will be used later in proving Theorem 3.10.

Lemma 3.6. *Modifying Algorithm 1 to determine whether G has a k-plex of size τ runs in $O(C \times (k+1)^{n-\tau})$ and $O^*((k+1)^{n-\tau})$ time.*

Proof. Let $\mathcal{T}$ be the search tree of running the revised Algorithm 1 that terminates a Branch&Bound instance (g, k, S, P) once $|V(g)|$ reaches τ. It suffices to prove that the number of leaf nodes in $\mathcal{T}$ rooted at any node I is at most $(k+1)^{|V(I.g)|-\tau}$. The proof follows a similar structure and argument as that of Lemma 3.4. If I is a leaf node, it is trivial that $\ell(I) = 1 \leq (k+1)^{|V(I.g)|-\tau}$ since $|V(I.g)| \geq \tau$. For a non-leaf node $I_0 = (g_0, S_0)$ (satisfying $|V(g_0)| > \tau$), assuming that $\ell(I') \leq (k+1)^{|V(I'.g)|-\tau}$ holds for every proper descendant I' of I_0, we in the following prove that $\ell(I_0) \leq (k+1)^{|V(g_0)|-\tau}$. To prove this, let's consider the path $(I_0, I_1, \ldots, I_q)$ that starts from I_0, always visits left child and stops once reaching a node $I_q \neq I_0$ satisfying $|V(g_q)| \leq |V(g_{q-1})| - 1$; recall that $I_i = (g_i, S_i)$ for $0 \leq i \leq q$. This means that $V(g_i) = V(g_0)$ holds for $1 \leq i < q$. It is trivial that $\ell(I_0) = \ell(I_{q+1}) + \ell(I_{q+2}) + \cdots + \ell(I_{2q}) + \ell(I_q) \leq (q+1) \times (k+1)^{|V(g_0)|-1-\tau}$; see Figure 3(a) for the nodes $\{I_{q+1}, \ldots, I_{2q}\}$. What remains to be proved is $q \leq k$. We prove this by considering three cases.

Case-I: $u_0 \in S_0$ (the case of Line 14) or $d_{S_0}(u_0) < |S_0|$ (the first sub-case of Line 15). Recall that u_0 is the vertex chosen at Line 13 for $I_0 = (g_0, S_0)$, and δ_i denotes the minimum value among the degrees, computed in S_i, of all vertices of g_i that have at least k non-neighbors in g_i; that is, $\delta_i = \min_{v \in V(g_i): |\overline{N}_{g_i}(v)| \geq k} d_{S_i}(v)$. Following the same argument as in Case-I of the proof of Lemma 3.4, it holds that for $1 \leq i \leq q-1$, $\delta_i = \delta_0$ and there is a vertex $v_i \in S_i$ satisfying $|\overline{N}_{g_i}(v_i)| \geq k$ and $d_{S_i}(v_i) = \delta_i = \delta_0$. According to the definition of q, **RR1** has no effect on I_{q-1} and thus we can get the same conclusion as Equation (1). Consequently $q \leq k-1$.

Case-II: $u_0 \notin S_0$, $d_{S_0}(u_0) = |S_0|$ and $|\overline{N}_{g_0}(u_0)| > k$ (the second sub-case of Line 15). Then, in $I_1 = (g_1, S_1)$, we have $u_0 \in S_1$ and $|\overline{N}_{g_1}(u_0)| > k$; thus, in I_1, we have either $u_1 \in S_1$ or $d_{S_1}(u_1) < |S_1|$. Consequently, we reach Case-I for I_1 and thus $q \leq k$.

Case-III: $u_0 \notin S_0$, $d_{S_0}(u_0) = |S_0|$ and $|\overline{N}_{g_0}(u_0)| = k$ (the case of Line 16). For the vertex w chosen at Line 17, we consider two subcases. **Subcase-1:** $|\overline{N}_g(w)| = 1$, corresponding to Line 18. Then, the unique non-neighbor v of w must satisfy $|\overline{N}_g(v)| \geq k$. Consequently, in $I_1 = (g_1, S_1)$, we have either $u_1 \in S_1$ or $d_{S_1}(u_1) < |S_1|$; that is, we reach Case-I for I_1, and thus $q \leq k$. **Subcase-2:** $|\overline{N}_g(w)| > 1$, corresponding to Line 19. Then, the branching vertex b_0 selected for I_0 satisfies $|\overline{N}_{g_0}(b_0)| \geq k$. Consequently, in $I_1 = (g_1, S_1)$, we have either $u_1 \in S_1$ or $d_{S_1}(u_1) < |S_1|$; that is, we reach Case-I for I_1, and thus $q \leq k$. □

3.2 kPlexT for Improving the Exponent of the Time Complexity

In this subsection, we propose a new framework for improving the exponent of the time complexity. The general idea is based on the following lemma.

Lemma 3.7. *[41] Any two non-adjacent vertices in a k-plex of size $2k-1$ or larger must have common neighbors.*

That is, the diameter of any k-plex of size $2k-1$ or larger is at most 2. Thus, if we know that the maximum k-plex size is at least $2k-1$, then for a branch-and-bound search instance (g, S) with $S \neq \emptyset$, we can dramatically reduce its size by removing from g all vertices whose shortest distance to a vertex $u \in S$ is greater than 2. However, in practice, we do not know in advance whether the maximum k-plex size is at least $2k-1$ or not. To resolve this issue, we propose a two-stage approach kPlexT by separating the search for k-plex of size at least $2k-1$ from the search for k-plex of size at most $2k-2$. That is, we first, at Stage-I, search for a maximum k-plex of size at least $2k-1$. If the k-plex found in Stage-I is of size at least $2k-1$, then it is guaranteed to be a maximum k-plex of the input graph G and we can terminate the algorithm. Otherwise, we go into Stage-II and search the input graph again for a maximum k-clique that is of size at most $2k-2$. We will show shortly that the exponent of the time complexity is significantly reduced.

Algorithm 2: kPlexT(G, k)

Input: A graph $G = (V, E)$ and an integer $k \geq 2$
Output: A maximum k-plex in G

1 $P \leftarrow \emptyset$;
/* Stage-I */
2 Let $(v_1, \ldots, v_n)$ be a degeneracy ordering of the vertices of G;
3 **for each** $v_i \in V(G)$ **do**
4 Let A be v_i's neighbors that are in $\{v_{i+1}, \ldots, v_n\}$, i.e., $A \leftarrow N(v_i) \cap \{v_{i+1}, \ldots, v_n\}$;
5 Let g be the subgraph of G induced by $N[A] \cap \{v_i, \ldots, v_n\}$;
6 Branch&Bound$(g, k, \{v_i\}, P)$;
/* Stage-II */
7 **if** $|P| < 2k - 2$ **then** Branch&Bound$(G, k, \emptyset, P)$;
8 **return** P;

The pseudocode of our two-stage approach kPlexT is shown in Algorithm 2. P records the currently found largest k-plex and is initialized as $\emptyset$ (Line 1). In Stage-I, we first compute a ***degeneracy ordering*** of the vertices of G and let it be $(v_1, \ldots, v_n)$ (Line 2); that is, v_i has the smallest degree in $G[\{v_i, \ldots, v_n\}]$ for each $1 \leq i \leq n$. Then, we iteratively process each vertex of G (Line 3), and conduct the following operations when processing vertex v_i:

(1) Get the set A of v_i's neighbors that are in $\{v_{i+1}, \ldots, v_n\}$, i.e., $A = N(v_i) \cap \{v_{i+1}, \ldots, v_n\}$ (Line 4).
(2) Extract the subgraph g of G induced by $N[A] \cap \{v_i, \ldots, v_n\}$, i.e., A and A's neighbors that are in $\{v_i, v_{i+1}, \ldots, v_n\}$ (Line 5).
(3) Compute a maximum k-plex that is in g and contains v_i by invoking Branch&Bound$(g, k, \{v_i\}, P)$ of Algorithm 1 (Line 6).

If the largest k-plex found in Stage-I is smaller than $2k - 2$, then we go into Stage-II which invokes Branch&Bound$(G, k, \emptyset, P)$ (Line 7). Note that, although we at Line 6 invoke Branch&Bound with a non-empty initial S, Lines 2–6 will find the maximum k-plex if its size is at least $2k - 1$: specifically, let v^* be the one that ranks the lowest according to the degeneracy ordering among all vertices of a maximum k-plex P^*, then Line 6 for $v_i = v^*$ will find P^*.

Time Complexity Analysis of kPlexT**.** Let Δ be the maximum degree of G, α be the degeneracy of G which is at most $O(\sqrt{m})$ and is small in practice [15], and $\omega_k(G)$ be the maximum k-plex size of G. The following time complexity analysis of kPlexT directly follows from Lemma 3.5.

Lemma 3.8. *Given a graph G and an integer k,* kPlexT *runs in $O(n \times (\alpha\Delta)^2 \times \gamma_k^{\alpha\Delta})$ time if $\omega_k(G) \geq 2k - 1$, and otherwise runs in $O\big(n \times (\alpha\Delta)^2 \times \gamma_k^{\alpha\Delta} + m \times \gamma_k^n\big)$ time.*

Proof. We first bound $|V(g)|$ for the subgraph g extracted at Line 5 of Algorithm 2. The definition of degeneracy ordering ensures that $|A| \leq \alpha$. Thus, $|N[A]| \leq |A| + |A| \times (\Delta - 1) + 1 \leq \alpha\Delta + 1$; the first inequality follows from the fact that v_i is a common neighbor of A. Consequently, each invocation of Branch&Bound at Line 6 takes time $O(|E(g)| \times \gamma_k^{|V(g)|-1}) \subseteq O((\alpha\Delta)^2 \times \gamma_k^{\alpha\Delta})$, since $|V(g)| \leq \alpha\Delta + 1$ and $|E(g)| \leq |V(g)|^2$. Hence, if the maximum k-plex size $\omega_k(G)$ is at least $2k - 1$, then kPlexT will terminate at Stage-I and thus run in $O(n \times (\alpha\Delta)^2 \times \gamma_k^{\alpha\Delta})$ time. Otherwise, invoking Branch&Bound$(G, k, \emptyset, P)$ at Line 7 runs in $O(m \times \gamma_k^n)$ time. Thus, the lemma holds. □

In the theorem below, we prove a tighter time complexity by using a more refined analysis of the search tree as shown in Figure 4.

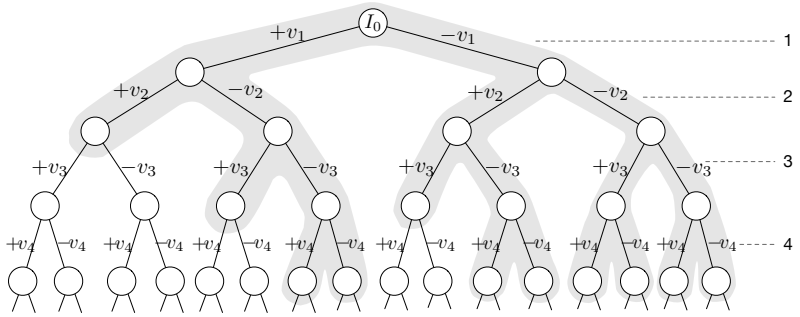


Fig. 4. Proof of Theorem 3.9

Theorem 3.9. *Given a graph G and an integer k,* kPlexT *runs in $O(n \times (\alpha\Delta)^{k+1} \times \gamma_k^{\alpha})$ time if $\omega_k(G) \geq 2k-1$, and in $O\big(n \times (\alpha\Delta)^{k+1} \times \gamma_k^{\alpha} + m \times \min\{\gamma_k^n, n^{2k-2}\}\big)$ time otherwise.*

Proof. Let f be the vertex obtained at Line 3 of Algorithm 2 and g be the subgraph extracted at Line 5; note that $d_g(f) = |A| \leq \alpha$. Let's consider the search tree $\mathcal{T}$, shown in Figure 4, of running Branch&Bound$(g, k, \{f\}, P)$. Let $I_0 = (g_0, S_0)$ be the root of $\mathcal{T}$; note that, $f \in S_0$ and $d_{g_0}(f) \leq \alpha$. In the following, we prove that the number of leaf nodes of $\mathcal{T}$ is $O((\alpha\Delta)^{k-1} \times \gamma_k^{\alpha})$.

To this end, we first prove that $|I_q| \leq d_{g_0}(f)$ holds for any node $I_q \in \mathcal{T}$ that has exactly $k-1$ *positive* edges (i.e., labeled as "+" in Figure 4) on the path from I_0 to I_q; note that $q \geq k-1$. Let $(I_0, I_1, \ldots, I_{q-1}, I_q)$ be the path from I_0 to I_q. If f has at most $k-1$ non-neighbors in g_0, then $|I_0| \leq |V(g_0)| - 1 \leq d_{g_0}(f) + k - 1$ and $|I_q| \leq |I_0| - q \leq d_{g_0}(f)$. In the following, we consider the case that f has at least k non-neighbors in g_0. If all the vertices that are added to S_q along the positive edges of the path $(I_0, \ldots, I_q)$ are non-neighbors of f, then **RR1** ensures that all vertices of $V(g_q) \setminus S_q$ are neighbors of f and thus $|I_q| = |V(g_q) \setminus S_q| \leq d_{g_0}(f)$. Otherwise, let I_i be the first node on the path from I_0 to I_q such that (i) the edge from I_i to I_{i+1} is a positive edge and (ii) the vertex added to S through this edge is a neighbor of f. This means that f is not selected as u at Line 13 of Algorithm 1. There are two cases depending on whether $|\overline{N}_{g_i}(f)| \geq k$. **Case-1:** f has at most $k-1$ non-neighbors in g_i. Let h $(\leq i)$ be the number of positive edges on the path from I_0 to I_i. Then, $|S_i| \geq 1 + h$, $|V(g_i)| \leq d_{g_i}(f) + k \leq d_{g_0}(f) + k$, and thus $|I_q| \leq |I_i| - (q - i) \leq |I_i| - (k - 1 - h) \leq d_{g_0}(f) + k - (h+1) - (k-1-h) = d_{g_0}(f)$. **Case-2:** f has at least k non-neighbors in g_i, but another vertex $u' \in V(g_i)$ is selected as u at Line 13 of Algorithm 1. Note that we must have $d_{S_i}(u') = d_{S_i}(f)$, since (1) every vertex added to S by reduction rule **RR2** must be neighbors of both u' and f and (2) we only apply reduction rule **RR3** to the branching vertex (i.e., applied to Line 10 of Algorithm 1 when b is removed from g); thus, **RR3** has no effect on the path from I_0 to I_i. Thus, $d_{g_i}(u') \leq d_{g_i}(f) \leq d_{g_0}(f)$. As $u' \in S_{i+1}$ and $d_{g_{i+1}}(u') \leq d_{g_i}(u') \leq d_{g_0}(f)$, we can continue the proof by considering u' as the new f and I_{i+1} as the new root.

Let t be the number of non-neighbors of f in g; note that $t \leq \alpha(\Delta - 1)$. 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 t total edges or $k-1$ positive edges; see the shaded subtree in Figure 4 for an illustration where $t = 4$ and $k = 3$. Let $\mathcal{L}$ be the set of leaf nodes of this subtree. Then, the number of leaf nodes of $\mathcal{T}$ is $\ell(I_0) \leq \sum_{I \in \mathcal{L}} \ell(I) \leq |\mathcal{L}| \times \gamma_k^{\alpha}$; the last inequality follows from the fact hat $|I| \leq \alpha$ for any $I \in \mathcal{L}$. 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 (e.g., see Figure 4), 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 $k-1$ of $\{1, 2, \ldots, t\}$. Consequently, $|\mathcal{L}| \leq \sum_{i=0}^{k-1} \binom{t}{i} \leq c \times t^{k-1} \leq c \times (\alpha\Delta)^{k-1}$ for a constant c [25].

Similarly, the time complexity of Stage-II can also be bounded by $O(m \times n^{2k-2})$ as the maximum k-plex size in Stage-II is guaranteed to be at most $2k-2$. This completes the proof. □

It is well known that $\alpha + k$ is an upper bound of $\omega_k(G)$. Concurrent to our work, Wang et al. [35] proposed the kPlex-gap algorithm and analyzed its time complexity by using the degeneracy gap (i.e., the difference between the degeneracy-based upper bound $\alpha + k$ and the maximum k-plex size $\omega_k(G)$) parameterization. Specifically, kPlex-gap runs in $O^*((\alpha\Delta)^{k+1}(k+1)^{\alpha+k-\omega_k(G)})$ time when $\omega_k(G) \geq 2k-1$, where the second term $(k+1)^{\alpha+k-\omega_k(G)}$ is small when $\alpha + k$ is close to $\omega_k(G)$. However, the time complexity is worse than $O^*((\alpha\Delta)^{k+1}2^{\alpha})$ when $(\alpha + k - \omega_k(G)) \log_2(k+1) > \alpha$ which is the case for many of our tested graphs. Nevertheless, we prove in the theorem below that the time complexity of kPlexT is also bounded by $O^*((\alpha\Delta)^{k+1}(k+1)^{\alpha+k-\omega_k(G)})$ when $\omega_k(G) \geq 2k-1$.

Theorem 3.10. *Given a graph G and an integer k,* kPlexT *with slight modification runs in $O^*((\alpha\Delta)^{k+1} \times (k+1)^{\alpha+k-\omega_k(G)})$ time when $\omega_k(G) \geq 2k-1$.*

Proof. To find $\omega_k(G)$, we iteratively test whether G has a k-plex of size τ for $\tau = \{\alpha + k, \alpha + k - 1, \ldots\}$; it will stop after the testing of $\tau = \omega_k(G)$. To prove the theorem, we only need to show that kPlexT determines whether G has a k-plex of size τ for $\tau \geq \omega_k(G)$ in $O^*((\alpha\Delta)^{k+1} \times (k+1)^{\alpha+k-\tau})$ time. The proof follows a similar structure and argument as that of Theorem 3.9. That is, we still consider the search tree $\mathcal{T}$ as shown in Figure 4 of running Branch&Bound$(g, k, \{f\}, P)$ where f is the vertex obtained at Line 3 of Algorithm 2 and g is the subgraph extracted at Line 5, and consider the set $\mathcal{L}$ of leaf nodes of the subtree of $\mathcal{T}$ as defined in the proof of Theorem 3.9. Recall that the root $I_0 = (g_0, S_0)$ of $\mathcal{T}$ satisfies $f \in S_0$ and $d_{g_0}(f) \leq d_g(f) = |A| \leq \alpha$, and the number of leaf nodes of $\mathcal{T}$ satisfies $\ell(I_0) \leq \sum_{I \in \mathcal{L}} \ell(I)$. In the following, we prove that for each $I \in \mathcal{L}$, $|V(I.g)| \leq \alpha + k$; consequently, $\ell(I) \leq (k+1)^{\alpha+k-\tau}$ following from Lemma 3.6 and thus $\ell(I_0) \leq c \times (\alpha\Delta)^{k-1} \times (k+1)^{\alpha+k-\tau}$ for a constant c.

Let $I_q = (g_q, S_q)$ be an arbitrary node of $\mathcal{L}$, and $(I_0, I_1, \ldots, I_{q-1}, I_q)$ be the path from I_0 to I_q. We prove $|V(g_q)| \leq \alpha + k$ in two cases depending on whether there are $k-1$ positive edges on the path. Note that, if f has at most $k-1$ non-neighbors in g_0, then it is trivial that $|V(g_q)| \leq |V(g_0)| \leq \alpha + k$. Thus, in the following we *assume f has at least k non-neighbors in g_0*.

Case-I: There are $k-1$ positive edges on the path from I_0 to I_q; note that $q \geq k-1$. If all the vertices that are added to S_q along the positive edges of the path $(I_0, \ldots, I_q)$ are non-neighbors of f, then **RR1** ensures that all vertices of $V(g_q) \setminus S_q$ are neighbors of f and thus $|V(g_q)| \leq \alpha + k$ (note that f can have at most $k-1$ non-neighbors in S_q). Otherwise, let I_i be the first node on the path from I_0 to I_q such that (i) the edge from I_i to I_{i+1} is a positive edge and (ii) the vertex added to S through this edge is a neighbor of f. This means that f is not selected as u at Line 13 of Algorithm 1. We consider two subcases. **Subcase-1:** f has at most $k-1$ non-neighbors in g_i. Then $|V(g_q)| \leq |V(g_i)| \leq \alpha + k$. **Subcase-2:** f has at least k non-neighbors in g_i, but another vertex $u' \in V(g_i)$ is selected as u at Line 13 of Algorithm 1. As argued in the proof of Theorem 3.9, it holds that $d_{S_i}(u') = d_{S_i}(f)$ and $d_{g_i}(u') \leq d_{g_i}(f) \leq |V(g_i)| - k - 1$, and we can continue the proof by considering u' as the new f and I_{i+1} as the new root.

Case-II: There are less than $k-1$ positive edges on the path from I_0 to I_q. Then, $q = t \geq k$ where t is the number of non-neighbors of f in g. Thus, there are at least $t - (k-2)$ negative edges (i.e., labeled as "−" in Figure 4) on the path from I_0 to I_q, and consequently $|V(g_q)| \leq |V(g_0)| - (t - k + 2) \leq t + \alpha + 1 - (t - k + 2) = \alpha + k - 1$. □

Remarks. From the proofs of Theorems 3.9 and 3.10, we can see that $\alpha\Delta$ used in the time complexities can be replaced by a smaller term $\max_g |V(g)|$ where g ranges over all subgraphs extracted at Line 5 of Algorithm 2. But for simplicity, we use the upper bound $\alpha\Delta$ of $\max_g |V(g)|$ in the time complexities.

As far as we know, we are the first to formally propose the two-stage framework and analyze its time complexity. Although a similar idea to Stage-I of kPlexT has been used in kPlexS [10], FP [13] and ListPlex [36], ours is different in the following ways. Firstly, kPlexS, ListPlex and FP themselves can only handle graphs with maximum k-plex size at least $2k-1$, while kPlexT can handle all graphs regardless of the maximum k-plex size. Secondly, for graphs with maximum k-plex size at least $2k-1$, kPlexS, FP and ListPlex run in $O^*(2^{\alpha\Delta})$, $O^*(\beta_k^{\alpha\Delta})$ and $O^*((\alpha\Delta)^{k+1}\beta_k^{\alpha})$ time, respectively, while kPlexT runs in $O^*(\min\{\gamma_k^{\alpha\Delta}, (\alpha\Delta)^{k+1}\gamma_k^{\alpha}\})$ time with $\gamma_k < \beta_k < 2$. In particular, to achieve its time complexity, ListPlex first enumerates all subsets of v_i's non-neighbors in g and then for each such subset of size at most $k-1$, it only needs to further consider v_i's neighbors which are of quantity at most α. Explicitly and exhaustively enumerating all these $\binom{\alpha\Delta}{k-1}$ subsets of v_i's non-neighbors is time-consuming in practice; our preliminary empirical study shows that ListPlex runs slower than FP in practice due to this exhaustive enumeration. In contrast, our algorithm achieves a better time complexity without special treatment of v_i's non-neighbors. Thirdly, we also proved the time complexity of kPlexT when the maximum k-plex size is small. Since $O(n^{2k-2}) = O(\gamma_k^{(2k-2)\log n})$, kPlexT reduces the exponent of the time complexity.

Note that, from the practical perspective, even if kPlexT goes into Stage-II, it is expected to still run faster than directly invoking kPlexBB with inputs G and k. The reasons are two-fold.

- Stage-II starts with a large lower bound that is equal to the size of the largest k-plex found in Stage-I. The large lower bound can shrink the input graph G significantly based on the pruning techniques that will be introduced in Section 3.3.
- Stage-II of kPlexT can stop once a k-plex of size $2k-2$ is found. This could terminate the search earlier.

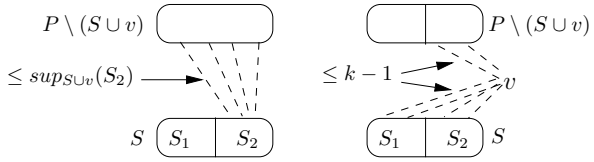
3.3 Techniques for Practical Performance

In the previous two subsections, we presented our algorithms by only concerning the theoretical time complexity. In this subsection, we consider techniques to improve the practical performance of kPlexT. Firstly, we adopt the widely used degree-based upper bound for pruning, i.e., the largest k-plex containing a vertex v is k plus the degree of v. Let lb be the size of the currently found largest k-plex in G. Then, given a branch-and-bound instance (g, S), if there is a vertex $v \in V(g) \setminus S$ such that $d_g(v) + k \leq lb$, we can remove v from g; if there is a vertex $u \in S$ such that $d_g(u) + k \leq lb$, we can prune the entire instance (g, S). In addition, we also propose a new reduction rule and a better initialization in the following.

A New Reduction Rule. Given an instance (g, S) and a vertex $v \in V(g) \setminus S$, we aim to compute an upper bound of the largest k-plex that is in g and contains $S \cup v$; if the computed upper bound is no larger than lb, then we can remove v from g. Our upper bound is based on two observations on the k-plexes that contain $S \cup v$, as shown in Figure 5. Let P be a k-plex containing $S \cup v$, and S_1 and S_2 be a partitioning of S such that $S_1 \cap S_2 = \emptyset$ and $S_1 \cup S_2 = S$. Firstly, let's consider the number of *non-edges* (i.e., missing edges) between S_2 and $P \setminus (S \cup v)$, see Figure 5(a). From S_2's viewpoint, it is at most

$$sup_{S\cup v}(S_2) := \sum_{u\in S_2} k - 1 - |\overline{N}_{S\cup v}(u)|$$

since each vertex u can have at most $k - 1 - |\overline{N}_{S\cup v}(u)|$ non-neighbors in $P \setminus (S \cup v)$. Thus, when we consider the non-edges from the viewpoint of $P \setminus (S \cup v)$, any k-plex P containing $S \cup v$ should



(a) Observation-I (b) Observation-II

Fig. 5. Two observations for k-plex

satisfy

$$\sum_{w\in P\setminus(S\cup v)} |\overline{N}_{S_2}(w)| \leq sup_{S\cup v}(S_2)$$

Let $(v_1, \ldots, v_{|V(g)\setminus(S\cup v)|})$ be the vertices of $V(g) \setminus (S \cup v)$ sorted in non-decreasing order with respect to $|\overline{N}_{S_2}(\cdot)|$. Then, the largest k-plex in g containing $S \cup v$ is of size at most

$$|S \cup v| + \max\left\{i : \sum_{1\leq j\leq i} |\overline{N}_{S_2}(v_j)| \leq sup_{S\cup v}(S_2)\right\} \tag{5}$$

Secondly, $P \setminus (S \cup v)$ can include at most $k - 1 - |\overline{N}_S(v)|$ non-neighbors of v, see Figure 5(b). Thus, in the upper bound computation via Equation (5), instead of considering all vertices of $V(g) \setminus (S \cup v)$, we can consider only all of v's neighbors and top-$(k - 1 - |\overline{N}_S(v)|)$ of v's non-neighbors that have the fewest non-neighbors in S_2. To get a tight upper bound from Equation (5), we set S_2 as $\{u \in S : |\overline{N}_g(u)| > k - 1\}$. The motivation is that every vertex of S_1 will always have at most $k - 1$ non-neighbors in any subset of $V(g)$ that contains S; thus, we can and should "ignore" the non-edges associated with S_1 which otherwise would contribute to $sup_{S\cup v}(S_2)$.

Algorithm 3: Reduce(g, S, lb)

```
1  S_2 ← {u ∈ S : |N̄_g(u)| > k − 1};
2  sup_S(S_2) ← Σ_{u∈S_2} k − 1 − |N̄_S(u)|;
3  for each v ∈ V(g) \ S do Compute |N̄_{S_2}(v)|;
4  Sort vertices of V(g) \ S in non-decreasing order regarding |N̄_{S_2}(·)|;
5  for each v ∈ V(g) \ S do
6      ub_v ← |S ∪ v|;
7      nn_v ← |N̄_S(v)|;
8      sup ← sup_S(S_2) − |N̄_{S_2}(v)|;
9      for each vertex w ∈ V(g) \ {S ∪ v} in non-decreasing order regarding |N̄_{S_2}(·)| do
10         if |N̄_{S_2}(w)| > sup or ub_v > lb then break;
11         if (v, w) ∉ E(g) and nn_v = k − 1 then continue;
12         if (v, w) ∉ E(g) then nn_v ← nn_v + 1;
13         sup ← sup − |N̄_{S_2}(w)|;
14         ub_v ← ub_v + 1;
15     if ub_v ≤ lb then Remove v from g;
```

The pseudocode implementing our new reduction rule is given in Algorithm 3. We first compute $sup_S(S_2)$ (Line 2) and sort all vertices of $V(g) \setminus S$ in non-decreasing order regarding $|\overline{N}_{S_2}(\cdot)|$ (Lines 3–4). Then, for each vertex $v \in V(g) \setminus S$ (Line 5), we compute an upper bound ub_v of the largest k-plex containing $S \cup v$ in g (Lines 6–14), and remove v from g if $ub_v \leq lb$ (Line 15). To compute the upper

bound, we first initialize ub_v as $|S \cup v|$ (Line 6) and the current number nn_v of non-neighbors of v as $|\overline{N}_S(v)|$ (Line 7), and initialize sup as $sup_{S \cup v}(S_2)$ which is obtained as $\sup_S(S_2) - |\overline{N}_{S_2}(v)|$ (Line 8). Then, we iteratively process each vertex $w \in V(g) \setminus (S \cup v)$ in non-decreasing order regarding $|\overline{N}_{S_2}(\cdot)|$ (Line 9); note that, sorting is not needed here since the vertices have already been sorted at Line 4. We count w towards the upper bound ub_v if (1) $sup \geq |\overline{N}_{S_2}(w)|$ and (2) w is a neighbor of v or $nn_v < k-1$. We update nn_v, sup and ub_v at Lines 12–14. It is easy to see that the worst-case time complexity of Algorithm 3 is $O(|V(g)|^2)$. Nevertheless, the algorithm usually runs very efficient in practice due to the pruning at Line 10.

A Better Initialization. For practical performance, it is usually desirable to initialize P with a large k-plex, rather than $\emptyset$, at Line 1 of Algorithm 2. Some of the existing works (e.g., [10, 19]) initialize P with the longest suffix of the degeneracy ordering that is a k-plex. This heuristic works reasonably well for most of the graphs, but it may produce a k-plex that is far from optimal for some graphs. To get a better initial k-plex, we propose to extract one subgraph g_v for each vertex $v \in V$, compute a large k-plex in g_v based on the degeneracy ordering of $V(g_v)$, and keep the largest one as the initial k-plex. To make the process also efficient, we extract g_v as the subgraph of G induced by v's higher-ranked neighbors according to the degeneracy ordering; this ensures that $|V(g_v)| \leq \alpha$ where α is the degeneracy of G. The time complexity of computing the initial k-plex is $O(\sum_{v \in V} |V(g_v)|^2) = O(\alpha m)$.

After obtaining an initial k-plex, we use its size to shrink the input graph G in the same way as [10]. That is, let lb be the size of the initial k-plex, we iteratively remove from G all vertices whose degrees are at most $lb - k$ and all edges that participate in less than $\max\{lb + 1 - 2k, 0\}$ triangles.

4 EXPERIMENTS

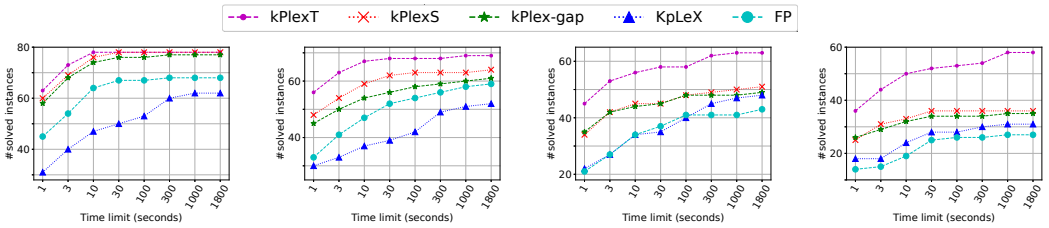
In this section, we evaluate the practical performance of our algorithm kPlexT [2] against the following existing algorithms for finding the maximum k-plex.

- KpLeX [3]: the existing algorithm proposed in [19].
- kPlexS [4]: the existing algorithm proposed in [10].
- kPlex-gap [5]: the existing algorithm proposed in [35].
- FP [6]: the existing algorithm proposed in [13] for enumerating all maximal k-plexes that are of size at least τ ($\geq 2k-1$).

We optimized the code of FP for computing the maximum k-plex by initializing τ as $2k-1$ and then keeping updating τ to be one plus the size of the currently found largest k-plex. This is exactly the strategy followed by kPlex-gap, while kPlexT, kPlexS and KpLeX differ slightly by initializing τ to be one plus the size of a heuristically found k-plex. By a similar argument to our proof of Theorem 3.9, one can show that FP runs in $O^*((\alpha\Delta)^{k+1}\beta_k^{\alpha})$ time; this is better than the proved time complexities of kPlexS and KpLeX. In addition, FP also incorporated advanced pruning techniques (e.g., upper bound-based pruning) based on the size threshold τ for improving the practical performance. Thus, we include FP as a baseline method in the experiments.

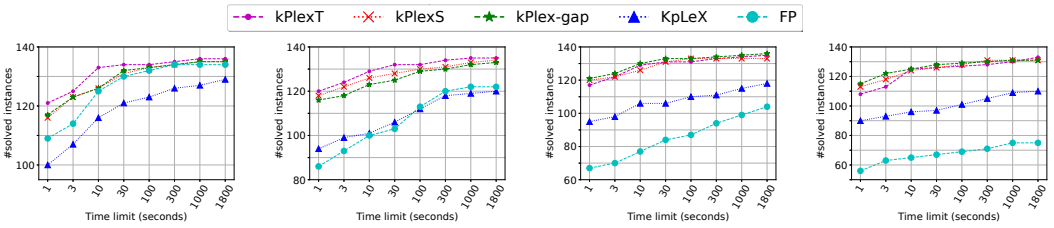
All the algorithms are implemented in C++ and compiled with -O3 flag. All experiments are run on a machine with an Intel Core i9 CPU and 64GB main memory, with parallelization disabled.

[2]The source code of kPlexT is released at https://lijunchang.github.io/Maximum-kPlex-v2/
[3]https://github.com/huajiang-ynu/kplex
[4]https://lijunchang.github.io/Maximum-kPlex/
[5]https://github.com/joey001/kplex_degen_gap
[6]https://github.com/qq-dai/kplexEnum



(a) $k = 3$ (b) $k = 5$ (c) $k = 7$ (d) $k = 10$

Fig. 6. Against existing algorithms on 10th DIMACS graphs (vary time limit, best viewed in color)



(a) $k = 3$ (b) $k = 5$ (c) $k = 7$ (d) $k = 10$

Fig. 7. Against existing algorithms on real-world graphs (vary time limit, best viewed in color)

Datasets. Same as [10, 17, 19, 41], we run the algorithms on the following two graph collections containing 223 graphs in total.

- The **10th DIMACS graphs** collection [7] contains 84 graphs with up to 5.09×10^7 vertices from the 10th DIMACS implementation challenge.
- The **real-world graphs** collection [8] contains 139 real-world graphs with up to 5.87×10^7 vertices from the Network Data Repository.

Metric. We record the total *processing time* of running an algorithm on a graph instance. 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 [10], we set a timeout of 1800s for each execution. We choose k from $\{3, 5, 7, 10\}$.

kPlexS, kPlex-gap and FP by design only work for graphs whose maximum k-plexes are of size at least $2k - 1$. Thus, in the experiment, if kPlex-gap and FP (resp. kPlexS) report a maximum k-plex of size less than $2k - 1$ (resp. $2k - 2$), then its reported time also includes the processing time of KpLeX. Note that, if the maximum k-plex reported by kPlexS is of size $2k - 2$, then it is guaranteed to be maximum [10]; but this does not hold for kPlex-gap and FP.

4.1 Against Existing Algorithms

We first evaluate the efficiency of kPlexT against the existing algorithms KpLeX, kPlexS, kPlex-gap and FP by comparing the number of graph instances that are solved by an algorithm within a specific time limit. The results on the 10th DIMACS graphs collection for $k = 3, 5, 7, 10$ are reported in Figure 6. We can see that our algorithm kPlexT outperforms all existing algorithms for all

[7] https://networkrepository.com/dimacs10.php
[8] http://lcs.ios.ac.cn/~caisw/Resource/realworld%20graphs.tar.gz

Table 3. Statistics of the solved instances (Columns 2, 5: #instances solved by at least one of the algorithms; Columns 3, 6: #instances where $\omega_k(G) < 2k-1$; Columns 4, 7: #instances where $\omega_k(G) \geq 2k-1$ and $2^\alpha \leq (k+1)^{\alpha+k-\omega_k(G)}$)

	DIMACS10			Real-world		
k	#solved	#small	2^α better	#solved	#small	2^α better
3	78	2	5	137	12	17
5	70	26	7	137	24	23
7	66	32	6	136	32	22
10	58	32	5	136	42	26

tested k values, and by a large margin for $k \geq 5$. For example, kPlexT solves 58 graphs within 1800s for $k = 10$, while the second-best performing algorithm kPlexS only solves 36 graphs. This demonstrates the practical efficiency of our algorithm. It can be observed that the speedup of kPlexT over kPlexS is higher when k is large. One of the reasons is that when k increases, the maximum k-plex size tends to be smaller than $2k-1$; see Column 3 of Table 3 for the exact number of instances with $\omega_k(G) < 2k-1$, among all solved instances as shown in Column 2. Consequently, the diameter-two-based pruning technique can no longer be applied, and the performance of kPlexS degrades significantly. Our algorithm kPlexT can still solve many of these instances based on our two-stage framework, since Stage-I, although may not find the maximum k-plex, can find a close to optimal solution and thus speed up the computation of Stage-II. In addition, we also show in Column 4 of Table 3 the number of instances where $\omega_k(G) \geq 2k-1$ and $2^\alpha \leq (k+1)^{\alpha+k-\omega_k(G)}$ (i.e., kPlexT has a lower time complexity than kPlex-gap) among the solved instances. It is also interesting to observe that FP, although originally designed for enumerating all large maximal k-plexes, runs faster than KpLeX for $k \leq 5$. This is because FP (1) exploits the diameter-two property of large k-plexes that is not utilized in KpLeX, and (2) also incorporates pruning techniques based on the size lower bound τ. Nevertheless, FP runs slower than KpLeX for $k \geq 7$, and slower than kPlexS and kPlex-gap for all k values. This indicates that the existing algorithms do not scale to relatively large k for enumerating all large maximal k-plexes, by noting that FP will run even slower for enumerating all large maximal k-plexes. We anticipate that incorporating some of the techniques from maximum k-plex solvers (including ours) could potentially improve the practical performance of enumerating all large maximal k-plexes. Nevertheless, a thorough investigation of this is beyond the scope of this paper, and we leave it to our future work.

To get a more detailed comparison of the algorithms, we report their actual processing time on the graph instances that have at least 10^6 vertices and are solved by at least one of the algorithms within 1800s. There are 26 such graphs in the 10th DIMACS graph collection. The results for $k = 5$ and $k = 7$ on these graphs are reported in Table 4 where "-" denotes that the algorithm does not finish within 1800 seconds; the number of vertices and edges of the graphs are reported in the second column and third column, respectively. In Table 4, we highlight in bold the processing time of an algorithm if it is within 1.5× the fastest running time among all algorithms for the same graph and the same k. Firstly, we observe that KpLeX times out on some graphs that have maximum k-plex size at least $2k-1$ (e.g., rows 2–7 and 21 for $k = 5$), while kPlexS and kPlex-gap can finish within seconds; we suspect that this is because the diameter-two property of large k-plexes can dramatically reduce the search space and thus make kPlexS and kPlex-gap run fast. Secondly, our algorithm kPlexT performs the best for most of the graphs. Nevertheless, there are also cases that KpLeX can efficiently find the maximum k-plex while kPlexT does not finish within 1800s, e.g., $k = 5$ on adaptive and $k = 7$ on hugetric-00010. This is because in these two cases, the initial k-plex

Table 4. Processing time of the algorithms on 10th DIMACS graphs that have at least 10^6 vertices for $k = 5$ and $k = 7$ ($\omega_k(G)$ is the maximum k-plex size, "-" means that the execution does not finish within 1800s, a running time is highlighted in bold if it is within 1.5× the fastest running time among all algorithms for the same graph and the same k, kPlex-g denotes kPlex-gap)

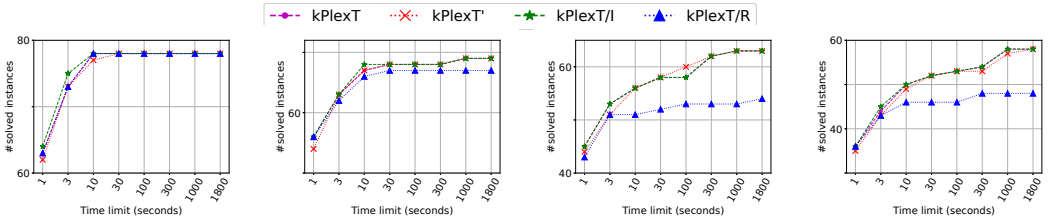
			$k = 5$					$k = 7$				
	n	m	$\omega_k(G)$	kPlexT	kPlexS	kPlex-g	KpLeX	$\omega_k(G)$	kPlexT	kPlexS	kPlex-g	KpLeX
adaptive	6M	13M	8	-	4.2	**0.38**	**0.26**	12	-	-	-	-
channel	4M	42M	10	**10**	**14**	**13**	-	12	**348**	644	-	-
delaunay_n20	1M	3M	9	**0.20**	0.67	0.51	-	12	**0.17**	-	-	-
delaunay_n21	2M	6M	9	**0.37**	0.75	0.83	-	12	**0.34**	-	-	-
delaunay_n22	4M	12M	9	**0.68**	1.3	1.1	-	12	**0.61**	-	-	-
delaunay_n23	8M	25M	9	**1.4**	2.8	2.6	-	12	**1.4**	-	-	-
delaunay_n24	16M	50M	9	**3.0**	5.0	4.7	-	12	**2.7**	-	-	-
hugetrace-00010	12M	18M	8	**8.2**	-	-	-	12	-	-	-	-
hugetrace-00020	16M	23M	8	-	-	-	-	12	**13**	-	-	-
hugetric-00010	6M	9M	8	-	-	-	-	12	-	4.2	**1.2**	**1.1**
hugetric-00020	7M	10M	8	-	-	-	-	12	**0.74**	2.0	1.4	1.3
inf-asia	11M	12M	8	**0.52**	-	-	-	12	**0.48**	1.2	0.89	0.73
inf-belgium	1M	1M	8	**0.10**	-	-	-	12	0.27	0.28	**0.08**	**0.07**
inf-europe	50M	54M	8	**5.3**	9.5	**4.9**	**4.1**	12	**2.7**	-	-	-
inf-germany	11M	12M	8	**1.8**	-	-	-	12	**0.73**	1.5	**1.0**	**0.83**
inf-great-britain	7M	8M	8	1.00	1.7	**0.56**	**0.46**	12	**0.39**	0.81	**0.55**	**0.45**
inf-italy	6M	7M	8	**0.48**	-	-	-	12	**0.25**	0.61	0.43	**0.36**
inf-netherlands	2M	2M	8	**0.24**	-	-	-	12	**0.08**	0.21	0.15	**0.12**
inf-road_central	14M	16M	8	**2.2**	3.6	-	-	12	**3.0**	6.8	**3.2**	**2.9**
inf-road_usa	23M	28M	8	**1.3**	**1.5**	-	-	12	**4.4**	8.4	**4.0**	**3.6**
packing	2M	17M	10	**6.1**	**7.6**	**6.9**	12	12	**126**	212	-	-
rgg_n_2_20_s0	1M	6M	20	**0.10**	**0.12**	**0.10**	0.24	23	**0.14**	**0.14**	**0.14**	0.25
rgg_n_2_21_s0	2M	14M	22	**0.15**	**0.17**	**0.15**	0.66	24	**0.21**	**0.22**	**0.22**	5.3
rgg_n_2_22_s0	4M	30M	23	**0.34**	**0.36**	**0.35**	1.5	25	**0.45**	**0.49**	**0.48**	6.4
rgg_n_2_23_s0	8M	63M	24	**0.77**	**0.86**	**0.81**	3.2	26	**1.0**	**1.1**	**1.1**	3.4
rgg_n_2_24_s0	16M	132M	25	**1.6**	**1.6**	**1.6**	7.5	27	**2.1**	**2.1**	**2.1**	7.5

found by KpLeX is equal to a computed upper bound of the maximum k-plex size; thus KpLeX can terminate immediately.

The number of instances solved by the algorithms on the real-world graphs collection by varying time limit is illustrated in Figure 7. The trend is generally similar to that for the 10th DIMACS graphs in Figure 6, i.e., kPlexT, kPlexS and kPlex-gap significantly outperform KpLeX and FP. But now the gap between kPlexT, kPlexS and kPlex-gap is much smaller; specifically, kPlexT outperforms kPlexS and kPlex-gap for $k \leq 5$, but performs similarly to kPlexS and kPlex-gap for $k \geq 7$. This is because graphs in this collection are relatively easy to solve and kPlexS is already very efficient for these graphs. For example, the number of graphs solved by kPlexS didn't drop significantly when k increases; out of 139 graphs, kPlexS solves 133 and 131 graphs within 1800s for $k = 7$ and $k = 10$, respectively. Thus, the room for practical improvement over kPlexS is not that much. Nevertheless, kPlexT solves two more instances than kPlexS does. The detailed processing time of the algorithms on the 15 graphs that have at least 10^6 vertices from the real-world graph collection is reported in Table 5. Our algorithm kPlexT generally performs the best. For example, kPlexT is at least three times faster than the next best algorithm on soc-lastfm, soc-pokec, socfb-A-anon for $k = 5$, and on inf-roadNet-CA, soc-pokec, socfb-A-anon for $k = 7$.

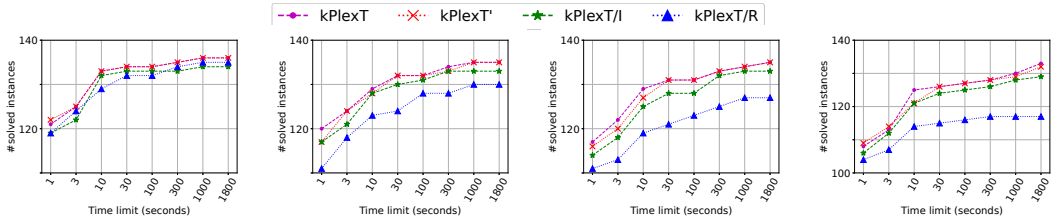
Table 5. Processing time of the algorithms on real-world graphs that have at least 10^6 vertices for $k = 5$ and $k = 7$ (kPlex-g denotes kPlex-gap)

			$k = 5$					$k = 7$				
	n	m	$\omega_k(G)$	kPlexT	kPlexS	kPlex-g	KpLeX	$\omega_k(G)$	kPlexT	kPlexS	kPlex-g	KpLeX
ca-hollywood	1M	56M	2209	**0.46**	**0.46**	**0.47**	**0.57**	2209	**0.47**	**0.51**	**0.48**	**0.67**
inf-road-usa	23M	28M	8	**1.4**	**1.3**	-	-	12	**4.1**	9.0	**4.2**	**3.8**
inf-roadNet-CA	1M	2M	8	**0.09**	**0.09**	-	-	12	**0.09**	0.45	0.39	0.36
inf-roadNet-PA	1M	1M	8	**0.05**	**0.05**	-	-	12	**0.05**	0.15	0.12	0.10
rt-retweet-crawl	1M	2M	17	0.17	0.17	**0.12**	**0.11**	20	**0.12**	0.19	**0.11**	**0.13**
soc-flixster	2M	7M	49	3.3	6.4	**0.80**	-	54	20	43	**4.0**	-
soc-lastfm	1M	4M	27	**5.3**	114	31	139	31	119	34	**6.2**	-
soc-livejournal	4M	27M	214	**0.69**	**0.72**	**0.71**	1.3	215	**0.79**	**0.95**	**0.81**	1.3
soc-pokec	1M	22M	34	**3.1**	9.6	9.7	12	39	**2.9**	9.6	9.7	77
soc-youtube-snap	1M	2M	26	**0.26**	**0.37**	**0.36**	44	30	1.1	**0.38**	**0.34**	1287
socfb-A-anon	3M	23M	37	**3.4**	12	12	185	41	**3.1**	11	11	-
socfb-B-anon	2M	20M	35	**17**	**15**	**18**	907	40	**7.2**	16	13	-
socfb-uci-uni	58M	92M	13	7.5	7.3	**3.4**	5.7	17	7.3	7.3	**2.1**	5.1
tech-as-skitter	1M	11M	75	0.58	**0.36**	**0.37**	-	78	23	0.73	**0.42**	-
web-wikipedia	1M	4M	32	**0.31**	**0.33**	**0.26**	126	32	0.38	0.45	**0.17**	-



(a) $k = 3$ (b) $k = 5$ (c) $k = 7$ (d) $k = 10$

Fig. 8. Ablation studies on 10th DIMACS graphs (vary time limit, best viewed in color)



(a) $k = 3$ (b) $k = 5$ (c) $k = 7$ (d) $k = 10$

Fig. 9. Ablation studies on real-world graphs (vary time limit, best viewed in color)

4.2 Ablation Study

In this subsection, we conduct ablation studies to evaluate the effectiveness of our reduction rules **RR2** and **RR3** in Section 3.1 as well as our two practical techniques proposed in Section 3.3. Specifically, we compare kPlexT with kPlexT′ which is kPlexT without **RR2** and **RR3**, kPlexT/R which is kPlexT without the reduction rule of Algorithm 3, and kPlexT/I which is kPlexT without the better initialization (i.e., kPlexT/I takes the longest suffix of the degeneracy ordering of the

input graph as the initial k-plex). The results on the number of graph instances that are solved by an algorithm within a specific time limit are shown in Figures 8 and 9. We can see that the better initialization method almost has no effect on the 10th DIMACS graphs, but all techniques have positive effect in improving the performance on the real-world graphs. By comparing Figure 8 with Figure 6, we can observe that kPlexT/R also outperforms the existing algorithms kPlexS, KpLeX and FP. The improvement is mainly achieved through our two-stage framework, where Stage-II takes the largest k-plex found in Stage-I as a lower bound and $2k-2$ as an upper bound.

5 EXTENSION TO OTHER PROBLEMS

In this section, we show that our improved time complexity presented in Theorem 3.9 carries over to other related problems.

Firstly, although the pseudocodes of Algorithms 1 and 2 are presented for finding the maximum k-plex, they can be straightforwardly extended to enumerate all maximal k-plexes with the same time complexity but possibly multiplying another polynomial factor for checking the maximality of an enumerated k-plex. Specifically, for Algorithm 1, we only need to change Line 6 to reporting g' if it is a maximal k-plex; this is because the three reduction rules **RR1**–**RR3** are all designed for maximal k-plex enumeration. Note that, to enumerate all maximal k-plexes regardless of their sizes, we would need to make the following changes to Algorithm 2: (i) in Stage-I, do not report maximal k-plexes of size smaller than $2k-1$ to avoid duplicates, and (ii) in Stage-II, always invoke Branch&Bound$(G, k, \emptyset)$ at Line 7 and terminate a branch-and-bound instance (g, S) once $|S| = 2k-2$. Thus, we can enumerate all maximal k-plexes of size at least $2k-1$ in $O^*((\alpha\Delta)^{k+1}\gamma_k^\alpha)$ time, and enumerate all maximal k-plexes in $O^*((\alpha\Delta)^{k+1}\gamma_k^\alpha + \min\{\gamma_k^n, n^{2k-2}\})$ time; this is better than the existing maximal k-plex enumeration algorithms, as summarized in Table 1. Also, we remark that Theorem 3.9 implies that the largest number of maximal k-plexes in a graph is $O^*((\alpha\Delta)^{k+1}\gamma_k^\alpha + \min\{\gamma_k^n, n^{2k-2}\})$, since each maximal k-plex must be a leaf in the search tree. Consequently, any polynomial-delay algorithm for maximal k-plex enumeration would run in this time and correspondingly, the time complexity of maximal k-plex enumeration over c-closed graphs (studied in [6]) and over c-weakly closed graphs (studied in [20]) is improved.

Secondly, our algorithm can be extended to enumerate all maximal λ-quasi-cliques in a graph for $\lambda \geq \frac{1}{2}$ in $O^*((\alpha\Delta)^{k+1}\gamma_k^\alpha)$ time. Given a real value λ, a graph g is a λ-quasi-clique if every vertex has at least $\lceil \lambda \cdot (|V(g)|-1) \rceil$ neighbors in g. As $|V(g)| - \lceil \lambda \cdot (|V(g)|-1) \rceil = \lfloor (1-\lambda)|V(g)| + \lambda \rfloor$, a λ-quasi-clique g is a $\lfloor (1-\lambda)|V(g)| + \lambda \rfloor$-plex. Moreover, we have the following lemma connecting maximal λ-quasi-cliques with maximal k-plexes; we omit its proof due to space limitation.

Lemma 5.1. *Any maximal λ-quasi-clique must be a a maximal k-plex for some $k \leq \lfloor (1-\lambda)\cdot ub + \lambda \rfloor$ where ub is an upper bound of the maximum λ-quasi-clique size.*

Thus, we can first enumerate all maximal k-plexes for $1 \leq k \leq \lfloor (1-\lambda)\cdot ub + \lambda \rfloor$, and then filter out those maximal k-plexes that are not maximal λ-quasi-cliques. It is known that the diameter of a λ-quasi-clique for $\lambda \geq \frac{1}{2}$ is at most 2 [27]. Therefore, we can invoke Stage-I of kPlexT to enumerate all diameter-two k-plexes. As γ_k is increasing with k, the time complexity is $O^*((\alpha\Delta)^{k+1}\gamma_k^\alpha)$ where $k = \lfloor (1-\lambda)\cdot ub + \lambda \rfloor$; this is better than the state-of-the-art time complexity $O^*(\sigma_k^{\alpha\Delta})$ achieved in [40] by noting that $\gamma_k < \sigma_k$.

Thirdly, our algorithm can be extended to enumerate all maximal $(k-1)$-biplexes over bipartite graphs in $O^*(\gamma_k^n)$ time; a $(k-1)$-biplex allows each vertex to have up-to $k-1$ non-neighbors, in the same way as a k-plex does, on the other side of the bipartite graph. Specifically, to enumerate all maximal $(k-1)$-biplexes, we "virtually" consider all vertices in the same side of the bipartite graph to be fully connected to each other (i.e., they form a clique), then enumerate all maximal k-plexes in this virtually transformed graph, and finally filter out those results that only contain vertices from

the same side. Note that, in the implementation, we do not actually add edges between vertices from the same side which would blow up the number of edges; instead, we only need to make use of this fact. We remark that, although Yu and Long [39] claimed that their algorithm enumerates all maximal $(k-1)$-biplexes also in $O^*(\gamma_k^n)$ time, there is a gap in their time complexity analysis due to simultaneously using two different analysis techniques. Let's illustrate this in the context of maximal k-plex enumeration and consider an instance $I = (g, S)$. The first analysis technique measures the size of I as $|I| = |V(g) \setminus S|$; this is what we used in this paper. The second analysis technique is the advanced measure-and-conquer technique of [16], which does not consider all vertices of $V(g) \setminus S$ to be of equal importance. Specifically, for some special nodes I_s in the search tree, [39] measures its size as $\|I_s\| = c \times n_{<k} + n_{=k}$ for a carefully chosen constant $0 < c < 1$; here, $n_{<k}$ and $n_{=k}$ are the number of vertices of $V(I_s.g) \setminus I_s.S$ that have less than k and exactly k non-neighbors in $I_s.g$, respectively. However, it is possible that a child I of I_s does not fall into this special category and its size is measured by $|I|$, and then we cannot replace $\gamma^{|I|}$ with $\gamma^{\|I\|}$ when analyzing the time complexity of I_s. This to some extent demonstrates that our techniques for reducing the base of the exponential time complexity to γ_k are nontrivial.

6 CONCLUSION

In this paper, we proposed a new algorithm kPlexT that not only improves the theoretical time complexity but also runs faster than the existing algorithms in practice, for the problem of maximum k-plex computation. Our algorithm remains relatively simple, but its time complexity analysis is nontrivial. We also showed that our improved time complexity carries over to other related problems such as enumerating all maximal k-plexes, quasi-cliques, and k-biplexes. We remark that our proposed reduction rules are orthogonal to the existing algorithms, and the existing pruning techniques (e.g., the partition-based upper bound of KpLeX and the second-order pruning technique of kPlexS) can also be incorporated into our algorithm. One direction of future work is to incorporate these techniques to further improve kPlexT's practical performance. Another direction of future work is to implement the ideas presented in Section 5 for these related problems and test its practical performance.

ACKNOWLEDGMENTS

Lijun Chang is supported by the Australian Research Council Fundings of FT180100256 and DP220103731.

REFERENCES

[1] 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.

[2] 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.

[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] Vladimir Batagelj and Matjaz Zaversnik. 2003. An O(m) Algorithm for Cores Decomposition of Networks. *CoRR* cs.DS/0310049 (2003).

[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] Balaram Behera, Edin Husic, Shweta Jain, Tim Roughgarden, and C. Seshadhri. 2022. FPT Algorithms for Finding Near-Cliques in c-Closed Graphs. In *Proc. of ITCS'22 (LIPIcs, Vol. 215).* 17:1–17:24.

[7] Devora Berlowitz, Sara Cohen, and Benny Kimelfeld. 2015. Efficient Enumeration of Maximal k-Plexes. In *Proc. of SIGMOD'15.* 431–444.

[8] Coenraad Bron and Joep Kerbosch. 1973. Finding All Cliques of an Undirected Graph (Algorithm 457). *Commun. ACM* 16, 9 (1973), 575–576.

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

[10] Lijun Chang, Mouyi Xu, and Darren Strash. 2022. Efficient Maximum k-Plex Computation over Large Sparse Graphs. *PVLDB* 16, 2 (2022), 127–139.

[11] Alessio Conte, Donatella Firmani, Caterina Mordente, Maurizio Patrignani, and Riccardo Torlone. 2017. Fast Enumeration of Large k-Plexes. In *Proc. of KDD'17*.

[12] Alessio Conte, Tiziano De Matteis, Daniele De Sensi, Roberto Grossi, Andrea Marino, and Luca Versari. 2018. D2K: Scalable Community Detection in Massive Networks via Small-Diameter k-Plexes. In *Proc. of KDD'18*. 1272–1281.

[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] Qiangqiang Dai, Rong-Hua Li, Xiaowei Ye, Meihao Liao, Weipeng Zhang, and Guoren Wang. 2023. Hereditary Cohesive Subgraphs Enumeration on Bipartite Graphs: The Power of Pivot-based Approaches. *Proc. ACM Manag. Data* 1, 2 (2023), 138:1–138:26.

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

[17] Jian Gao, Jiejiang Chen, Minghao Yin, Rong Chen, and Yiyuan Wang. 2018. An Exact Algorithm for Maximum k-Plexes in Massive Graphs. In *Proc. IJCAI'18*.

[18] Guimu Guo, Da Yan, M. Tamer Özsu, Zhe Jiang, and Jalal Khalil. 2020. Scalable Mining of Maximal Quasi-Cliques: An Algorithm-System Codesign Approach. *Proc. VLDB Endow.* 14, 4 (2020), 573–585.

[19] Hua Jiang, Dongming Zhu, Zhichao Xie, Shaowen Yao, and Zhang-Hua Fu. 2021. A New Upper Bound Based on Vertex Partitioning for the Maximum K-plex Problem. In *Proc. of IJCAI'21*. 1689–1696.

[20] Tomohiro Koana, Christian Komusiewicz, and Frank Sommer. 2020. Computing Dense and Sparse Subgraphs of Weakly Closed Graphs. In *Proc. of ISAAC'20 (LIPIcs, Vol. 181)*. 20:1–20:17.

[21] John M. Lewis and Mihalis Yannakakis. 1980. The Node-Deletion Problem for Hereditary Properties is NP-Complete. *J. Comput. Syst. Sci.* 20, 2 (1980), 219–230.

[22] 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.

[23] Can Lu, Jeffrey Xu Yu, Hao Wei, and Yikai Zhang. 2017. Finding the Maximum Clique in Massive Graphs. *PVLDB* 10, 11 (2017), 1538 – 1549.

[24] Benjamin McClosky and Illya V. Hicks. 2012. Combinatorial algorithms for the maximum k-plex problem. *J. Comb. Optim.* 23, 1 (2012), 29–49.

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

[26] Hannes Moser, Rolf Niedermeier, and Manuel Sorge. 2012. Exact combinatorial algorithms and experiments for finding maximum k-plexes. *J. Comb. Optim.* 24, 3 (2012), 347–373.

[27] Jian Pei, Daxin Jiang, and Aidong Zhang. 2005. On mining cross-graph quasi-cliques. In *Proc. of KDD'05*. 228–238.

[28] 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.

[29] S. Seidman and B. L. Foster. 1978. A graph-theoretic generalization of the clique concept. *Journal of Mathematical Sociology* 6 (1978), 139–154.

[30] 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.

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

[32] Charalampos Tsourakakis, Francesco Bonchi, Aristides Gionis, Francesco Gullo, and Maria Tsiarli. 2013. Denser than the densest subgraph: extracting optimal quasi-cliques with quality guarantees. In *Proc. of KDD'13*. 104–112.

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

[34] Zhuo Wang, Qun Chen, Boyi Hou, Bo Suo, Zhanhuai Li, Wei Pan, and Zachary G. Ives. 2017. Parallelizing maximal clique and k-plex enumeration over graph data. *J. Parallel Distributed Comput.* 106 (2017), 79–91.

[35] Zhengren Wang, Yi Zhou, Chunyu Luo, and Mingyu Xiao. 2023. A Fast Maximum k-Plex Algorithm Parameterized by the Degeneracy Gap. In *Proc. of IJCAI'23*. 5648–5656.

[36] Zhengren Wang, Yi Zhou, Mingyu Xiao, and Bakhadyr Khoussainov. 2022. Listing Maximal k-Plexes in Large Real-World Graphs. In *Proc. of WWW'22*. 1517–1527.

[37] Bin Wu and Xin Pei. 2007. A Parallel Algorithm for Enumerating All the Maximal k -Plexes. In *PAKDD Workshops'07*. 476–483.

[38] Mingyu Xiao, Weibo Lin, Yuanshun Dai, and Yifeng Zeng. 2017. A Fast Algorithm to Compute Maximum k-Plexes in Social Network Analysis. In *Proc. of AAAI'17*.
[39] Kaiqiang Yu and Cheng Long. 2023. Maximum k-Biplex Search on Bipartite Graphs: A Symmetric-BK Branching Approach. *Proc. ACM Manag. Data* 1, 1 (2023), 49:1–49:26.
[40] Kaiqiang Yu and Cheng Long. 2024. Fast Maximal Quasi-clique Enumeration: A Pruning and Branching Co-Design Approach. *Proc. ACM Manag. Data* 2 (2024).
[41] Yi Zhou, Shan Hu, Mingyu Xiao, and Zhang-Hua Fu. 2021. Improving Maximum k-plex Solver via Second-Order Reduction and Graph Color Bounding. In *Proc. of AAAI'21*. 12453–12460.
[42] 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.

Received July 2023; revised October 2023; accepted November 2023