Efficient Maximum k-Plex Computation over Large Sparse Graphs

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

Mouyi Xu
The University of Sydney
Australia
moxu7046@uni.sydney.edu.au

Darren Strash
Hamilton College
US
dstrash@hamilton.edu

ABSTRACT

The k-plex model is a relaxation of the clique model by allowing every vertex to miss up to k neighbors. Designing exact and efficient algorithms for computing the maximum k-plex in a graph has been receiving increasing interest recently. However, the existing algorithms are still inefficient due to having major limitations. We in this paper design a new algorithm kPlexS for the maximum k-plex problem, with three novel contributions. Firstly, we propose a new framework for computing maximum k-plex over large sparse graphs, by iteratively extracting small dense subgraphs from it and then solving each of the extracted dense subgraphs by a branch-and-bound search. Secondly, we propose an efficient reduction algorithm CTCP to reduce the input graph size by exhaustively conducting vertex reduction and edge reduction. CTCP computes a smaller reduced graph and also has a lower time complexity than the existing techniques. Moreover, we iteratively invoke CTCP to reduce the input graph once a vertex has been processed and removed from it. Thirdly, we develop a branch-and-bound algorithm BBMatrix specifically targeting the dense subgraphs that are extracted from the input graph. BBMatrix represents its input graph by an adjacency matrix, and utilizes both first-order (i.e., individual vertices) and second-order information (i.e., pairs of vertices) for reduction and upper bounding. In addition, incremental techniques are proposed to efficiently apply the reduction and upper bounding during the recursion. Extensive empirical studies on large real graphs demonstrate that our algorithm kPlexS outperforms the state-of-the-art algorithms BnB, Maplex, and KpLeX.

PVLDB Reference Format:
Lijun Chang, Mouyi Xu, and Darren Strash. Efficient Maximum k-Plex Computation over Large Sparse Graphs. PVLDB, 15(1): XXX-XXX, 2022.
doi:XX.XX/XXX.XX

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

1 INTRODUCTION

The graph model has been used in a wide range of data analysis applications such as social media, communication networks, collaboration networks, web graphs, and the Internet, as it naturally captures the relationship between entities. Graph data in these applications are usually globally sparse — e.g., the average degree is orders of magnitude smaller than the number of vertices — but locally dense [7]. Identifying locally dense (i.e., cohesive) subgraphs has many applications. For example, identifying large cohesive subgraphs in social networks has been used to detect money laundering and narcotics activity [1], and cohesive subgraphs are listed as instrumental to the detection of the social network of terrorists involved in the 2001 World Trade Centre terrorist attacks [13].

One classic notion of cohesive subgraph is *clique* which requires every pair of distinct vertices to be connected by an edge, and efficient algorithms for maximal clique enumeration and maximum clique computation have been designed, e.g., in [4, 6, 10, 16, 17, 27, 30]. Nonetheless, the clique concept is often too restrictive, as large and tightly connected communities in real networks hardly appear as cliques. In view of this, various clique relaxations have been formulated in the literature [25], such as k-plex, n-clan, n-club, and s-clique. *k-plex* relaxes the clique concept by allowing every vertex to miss up to k neighbors (including the vertex itself) in a subgraph; note that a 1-plex is a clique. Both the problem of maximum k-plex computation, which computes a k-plex with the largest number of vertices, and the problem of maximal k-plex enumeration, which reports all k-plexes that are maximal, have been receiving increasing interests recently, e.g., see [2, 3, 8, 9, 11, 12, 19, 20, 28, 33–35, 37, 38].

In this paper, we study the *maximum k-plex* problem. As the problem is NP-hard [14], the only viable option for designing exact algorithms is branch-and-bound search which runs in exponential time in the worst case. To speed up the computation, all of the three most recent approaches BnB [11], Maplex [38] and KpLeX [12] split the computation into two stages, by first computing a kernel graph in polynomial time in Stage-I and then process the much smaller kernel graph (than the input graph) by branch-and-bound search in Stage-II. Specifically, the ***kernel graph*** is a subgraph of the input graph $G = (V, E)$ obtained by removing unpromising vertices and/or edges, which are determined based on the size of a heuristically computed k-plex, from it. Although different techniques have been proposed in [11, 12, 38] for these two stages, they are inefficient due to having the following two major limitations.

(1) For Stage-I, the existing algorithms obtain a large kernel graph and also have a high time complexity for computing it; note that, different algorithms may obtain different kernel graphs, and in general the smaller the kernel graph the more efficient the branch-and-bound search in Stage-II. First, all existing works, except [38], conducts only vertex reduction. Second, although Maplex [38] also conducts edge reduction, it does not iteratively conduct the reductions until convergence due to its high time complexity; our empirical study shows that the size of the kernel graph obtained by Maplex [38] is the same as that by KpLeX [12]. Third, they have a high time complexity of either $O(r \cdot |V| \cdot |E|)$ [12]

Proceedings of the VLDB Endowment, Vol. 15, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

or $O(r \cdot |E|^{1.5})$ [38], where r is the number of iterations of applying the reductions which can be as large as $|V|$.

(2) For Stage-II, all the existing branch-and-bound search algorithms exploit only first-order information (i.e., degrees of individual vertices) for upper bounding and pruning, which are not as powerful as second-order information (i.e., common neighbors for pairs of vertices). Note that although the edge reduction used in Stage-I of Maplex [38] utilizes second-order information, it is used only in Stage-I but not Stage-II of Maplex, and moreover, non-adjacent vertex pairs have not been exploited.

We design a new algorithm kPlexS for efficient maximum k-plex computation over large sparse graphs, by developing novel techniques to resolve the above two limitations. Firstly, for Stage-I, we formalize the idea of *core-truss co-pruning* for computing a kernel graph that is guaranteed to be no larger than the kernel graphs obtained by the existing algorithms. That is, given a heuristically computed k-plex of size lb, we reduce the input graph G to its maximal subgraph that is both a $(lb+1-k)$-core — by iteratively removing vertices whose degrees are less than $lb+1-k$ (***vertex reduction***) — and a $(lb+1-2k)$-truss — by iteratively removing edges that participates in less than $lb+1-2k$ triangles (***edge reduction***). However, the naive approach of iteratively applying vertex reduction followed by edge reduction until convergence would take $O(r \cdot |E|^{1.5})$ time, where r is the number of iterations and can be as large as $|V|$. To speed up the computation, we design the CTCP algorithm to conduct core-truss co-pruning in $O(\delta(G) \times |E|)$ time, where $\delta(G)$ is the degeneracy of G that is small in practice and guaranteed to be at most $\sqrt{|E|}$ [10]. Consequently, kPlexS computes a smaller kernel graph in a faster time than the existing algorithms.

Secondly, for Stage-II, we propose to further exploit the second-order information, in addition to the first-order information, for upper bounding and pruning in the branch-and-bound search. To do so, given a partial solution (i.e., k-plex) S, for each pair of vertices u and v in the current working graph, we need to compute the number of common neighbors of u and v that are not in S, denoted $\mathsf{cn}_{\bar{S}}(u,v)$. We then utilize these $\mathsf{cn}_{\bar{S}}(\cdot,\cdot)$ values to obtain upper bounds such that we can prune either vertices from the graph (see our reduction rule **RR4** in Section 5), or edges from the graph (see our reduction rule **RR5** in Section 5), or even the entire search branch/instance (see our upper bound **UB2** in Section 5). However, it is inefficient to compute $\mathsf{cn}_{\bar{S}}(\cdot,\cdot)$ on-the-fly and is challenging, if not infeasible, to store and maintain $\mathsf{cn}_{\bar{S}}(\cdot,\cdot)$ for large sparse graphs; note that, the kernel graph computed from Stage-I could still be sparse, see Table 4 in Section 6. To circumvent this, we design a new framework for computing maximum k-plex over large sparse graphs, by running branch-and-bound search over *dense* subgraphs that are iteratively extracted from the input graph. Then, we develop a new branch-and-bound search algorithm BBMatrix specifically targeting these dense subgraphs. In BBMatrix, we represent its input graph by an adjacency matrix, and incrementally maintain $\mathsf{cn}_{\bar{S}}(\cdot,\cdot)$ for all vertex pairs and use these values for upper bounding and pruning.

Contributions. Our main contributions are as follows.

- We design a new framework for computing maximum k-plex over large sparse graphs, by running branch-and-bound search over dense subgraphs that are iteratively extracted from a sparse graph. (Section 3)
- We develop an efficient algorithm CTCP for computing the kernel graph. CTCP computes a smaller kernel graph and also has a lower time complexity than the existing algorithms. Moreover, CTCP is not only used in the preprocessing step to compute the kernel graph, but also iteratively invoked by our framework to reduce the graph size whenever a vertex has been processed and removed. (Section 4)
- We propose a branch-and-bound algorithm BBMatrix specifically targeting the dense subgraphs that are extracted from the input graph. BBMatrix uses adjacency matrix graph representation and exploits both first-order and second-order information for upper bounding and pruning. In addition, incremental techniques are proposed to efficiently apply the upper bounding and pruning. (Section 5)

We conduct extensive empirical studies on two collections of benchmark graphs (Section 6). The results show that for any given time limit, our algorithm kPlexS always solves more graph instances than the state-of-the-art algorithms BnB, Maplex, and KpLeX.

Related Works. We categorize related works as follows.

(1) Maximum k-plex Computation. The concept of k-plex was introduced by Seidman and Foster [28] in the context of social network analysis. The NP-hardness of maximum k-plex computation follows from the general proof of [14]. Balasundaram et al. [2] formulated an integer linear program for maximum k-plex computation, and designed a branch-and-cut implementation IPBC. McClosky and Hicks [19] developed a combinatorial algorithm OsterPlex which is an adaptation of the maximum clique computation algorithm proposed in [22]. Moser et al. [20] designed an algorithm GuidedBranching for the complement problem that aims to find the largest vertex-induced subgraph whose maximum degree is at most $k-1$. Xiao et al. [37] designed the BS algorithm that improves the worst-case running time to $c^n n^{O(1)}$ where $c < 2$ is a constant depending only on k. BnB [11], Maplex [38], and KpLeX [12] are the three most recent algorithms, which have been discussed in above.

(2) Maximal k-plex Enumeration. The problem of enumerating all maximal k-plexes (i.e., those k-plexes that are not contained in a larger k-plex) is also extensively studied. Most of the maximal k-plex enumeration algorithms, such as [9, 33, 35], follow the framework of the Bron-Kerbosch algorithm [4] that enumerates all maximal cliques. Besides these algorithms, a polynomial delay enumeration algorithm is proposed in [3], and simple pruning techniques are used in [8, 34] for enumerating maximal k-plexes that are larger than a given threshold τ. However, these enumeration algorithms are inefficient for computing maximum k-plex due to lack of advanced pruning and bounding techniques. On the other hand, the maximum k-plex computed by our algorithm provides a guideline for choosing the threshold τ.

(3) Maximum Clique Computation. Designing exact algorithms for maximum clique computation has also been extensively studied, e.g., [5, 6, 15, 16, 23, 24, 26, 27, 30, 31, 36]. Upper bounds based on graph coloring and MaxSAT reasoning have been shown to be the most successful techniques for maximum clique computation.

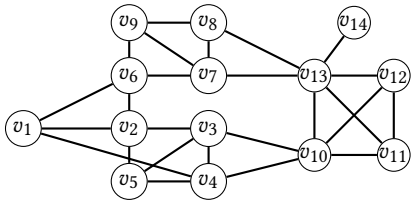


Figure 1: An example graph

However, they cannot be applied to compute maximum k-plex for $k \geq 2$, as they heavily rely on the clique property. On the other hand, second-order techniques have not been used in the existing studies of maximum clique computation. We remark that for the special case of $k = 1$ where a 1-plex is also a clique, the existing maximum clique solvers will be more efficient than our algorithms.

2 PRELIMINARIES

In this paper, we focus on a large *unweighted* and *undirected* graph $G = (V, E)$, where V is the set of vertices and E is the set of undirected edges. We consider only *simple* graphs, i.e., without self-loops and parallel edges. We denote the undirected edge between u and v by both (u, v) and (v, u); then, u (resp. v) is said to be adjacent to and a neighbor of v (resp. u). The set of neighbors of u in G is $N_G(u) = \{v \in V \mid (u, v) \in E\}$, and the *degree* of u in G is $d_G(u) = |N_G(u)|$. Given a vertex subset S of G, we use $G[S]$ to denote the subgraph of G induced by S, i.e., $G[S] = (S, \{(u, v) \in E \mid u, v \in S\})$. For ease of presentation, we simply refer to an unweighted and undirected graph as a graph, and omit the subscript G from the notations when the context is clear. For an arbitrary given graph g, we denote its set of vertices and its set of edges by $V(g)$ and $E(g)$, respectively.

Definition 2.1 (Clique). A graph g is a clique if there is an edge in g between every pair of distinct vertices, or equivalently, every vertex $u \in V(g)$ has degree $d_g(u) = |V(g)| - 1$.

Definition 2.2 (k-plex). A graph g is a k-plex if for every vertex $u \in V(g)$, its degree satisfies $d_g(u) \geq |V(g)| - k$, i.e., u misses edges to at most k vertices (including u itself).

The k-plex concept is a relaxation of the clique concept, and a 1-plex is a clique according to the definition. Obviously, if a subgraph g of G is a k-plex, then the subgraph $G[V(g)]$ of G induced by vertices $V(g)$ is also a k-plex. Thus, in this paper, *we simply refer to a k-plex by its set of vertices.* The *size of a k-plex $P \subseteq V$* is measured by its number of vertices, denoted $|P|$. A k-plex P of G is a *maximal k-plex* if every proper superset of P in G is not a k-plex. A k-plex P of G is a *maximum k-plex* if its size is the largest among all k-plexes of G; note that maximum k-plex is not unique. Consider the graph in Figure 1, $\{v_2, v_3, v_4, v_5\}$, $\{v_6, v_7, v_8, v_9\}$ and $\{v_{10}, v_{11}, v_{12}, v_{13}\}$ are three maximum 2-plexes of size 4.

For two vertices u and v that are not adjacent (i.e., not connected by an edge), we call v (resp. u) a *non-neighbor* of u (resp. v); note that a vertex is considered neither a neighbor nor a non-neighbor of itself. Then, in a k-plex P, every vertex has at least $|P| - k$ neighbors and equivalently at most $k - 1$ non-neighbors. The property of k-plex is *hereditary*, i.e., any subset of a k-plex is also a k-plex.

Problem Statement. Given a graph $G = (V, E)$ and an integer $k \geq 2$, in this paper we study the problem of maximum k-plex computation, which aims to find the largest k-plex in G that is of size at least $2k - 1$. If there is no k-plex of size at least $2k - 1$, then we report an arbitrary k-plex.

Our main motivations for only considering k-plexes of size at least $2k - 1$ are as follows.

- k is usually small in practice, e.g., k is set to be at most 5 in the empirical studies of [11, 37, 38]. Thus, $2k - 1$ is small, and it is natural to search for k-plexes of size at least $2k - 1$.
- A k-plex of size smaller than $2k - 1$ may be disconnected, e.g., any two disjoint $(k - 1)$-cliques form a k-plex of size $2k - 2$. In contrast, any k-plex of size at least $2k - 1$ is guaranteed to be connected.

When our algorithm reports a k-plex of size smaller than $2k - 2$, it means that the maximum k-plex size is at most $2k - 2$; we then can invoke any of the existing algorithms, if a maximum k-plex is still needed in this case. Note that, when our algorithm reports a k-plex of size $2k - 2$, then this k-plex is also guaranteed to be maximum.

Frequently used notations are summarized in Table 1.

Table 1: Frequently used notations

Notation	Meaning
$G = (V, E)$	an unweighted and undirected graph with vertex set V and edge set E
$P, S \subseteq V$	k-plexes
$\bar{S}$	the set of vertices that are outside S
$N_S(u)$	the set of neighbors of u that are in S
$d_S(u)$	the cardinality of $N_S(u)$, i.e., $d_S(u) = \|N_S(u)\|$
$N_G^{\leq 2}(u)$	the set of vertices that are at most 2-hops away from u in G, i.e., u's neighbors and u's neighbors' neighbors
$\mathrm{cn}_{\bar{S}}(u, v)$	the number of common neighbors of u and v in $\bar{S}$

2.1 k-Core and k-Truss

We review the concepts of k-core and k-truss here, which will be used in our pruning techniques in Section 4.

Definition 2.3 (k-core [29]). Given a graph G and an integer k, the k-core of G is the maximal subgraph g of G such that every vertex $u \in V(g)$ has degree $d_g(u) \geq k$ in the subgraph g.

k-core is a *vertex-induced* subgraph. For example, the entire graph in Figure 1 is a 1-core, and the graph obtained by excluding vertex v_{14} is a 3-core. Note that although every vertex in a k-plex P must have degree at least $|P| - k$ in P, the concepts of k-plex and k-core are inherently different. That is, the minimum degree requirement in k-plex depends on the size of the k-plex, while that in k-core is independent of its size. This makes maximum k-plex computation NP-hard [14], while computing (maximum) k-core is in P.

A related concept is the ***core number*** of a vertex u, denoted $\mathrm{core}(u)$, which is the largest k such that u is in the k-core. Given the core numbers of all vertices, the k-core is simply (the subgraph of G induced by) the set of vertices with core number at least k. The problem of computing the core number for all vertices is known as the ***core decomposition*** problem. It can be computed by the peeling algorithm in $O(|V| + |E|)$ time [18], which iteratively removes the vertex with the smallest degree from the graph. The

peeling sequence of vertices is known as the ***degeneracy ordering***, and the maximum core number among all vertices of G is known as the ***degeneracy*** of G, denoted $\delta(G)$, which is at most $\sqrt{|E|}$ [10].

Definition 2.4 (k-truss [32]). Given a graph G and an integer k, the k-truss of G is the maximal subgraph g of G such that every edge $(u, v) \in E(g)$ participates in at least k triangles, i.e., $|N_g(u) \cap N_g(v)| \geq k, \forall (u, v) \in E(g)$.[1]

k-truss is an *edge-induced* subgraph. For the graph in Figure 1, the subgraph obtained by excluding edges (v_1, v_4) and (v_{13}, v_{14}) is a 1-truss, i.e., each edge participates in at least one triangle. k-truss can be considered as a higher-order version of k-core. That is, each edge corresponds to a node, and each triangle corresponds to a hyper-edge. Hence, the k-truss can also be computed by the peeling algorithm, while the time complexity becomes $O(\delta(G) \times |E|)$ [32].

3 OUR FRAMEWORK

In this paper, we propose a new framework for maximum k-plex computation over large sparse graphs; note that our framework also works for dense graphs, and some of the graphs tested in our experiments in Section 6 are dense. Instead of directly conducting a branch-and-bound search on the input large sparse graph, we conduct branch-and-bound searches on dense subgraphs that are iteratively extracted from it. Our framework is mainly based on the observation of the following lemma.

Lemma 3.1. *[38] Any two non-adjacent vertices in a k-plex of size $\ell \geq 2k - 1$ must have at least $\ell - 2k + 2$ common neighbors.*

Following Lemma 3.1 we know that *any two non-adjacent vertices in a k-plex of size at least $2k - 1$ must have at least one common neighbor*; this also means that any k-plex of size $\geq 2k - 1$ must be connected. As a result, we can restrict the computation of the maximum k-plex containing a vertex u in G to the subgraph induced by vertices $N_G^{\leq 2}(u)$; recall that we are only interested in k-plexes of size at least $2k - 1$. Here, $N_G^{\leq 2}(u)$ denotes the set of vertices that are at most 2-hops away from u in G. For example, to search for a k-plex containing v_1 and of size at least $2k - 1$ for the graph in Figure 1, we can restrict the computation to the subgraph induced by vertices $N_G^{\leq 2}(v_1) = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_9, v_{10}\}$. The advantages of working with the subgraphs induced by $N_G^{\leq 2}(u)$ are twofold.

- The subgraphs are much smaller than the input graph.
- The subgraphs are dense as observed by our empirical studies (see Table 4 in Section 6), and thus enables second-order techniques which would otherwise be time-consuming to be applied on sparse graphs.

Our Framework. Based on the above observation, we propose to iteratively extract small dense subgraphs from G and then process these subgraphs by branch-and-bound search. Specifically, we iteratively conduct the following operations until the graph is empty:

(1) Choose a vertex u from the graph.
(2) Compute the maximum k-plex that contains u.
(3) Remove u from the graph.

[1]For presentation simplicity, the k-truss here is defined slight differently from the literature which requires every edge to participate in at least $k - 2$ triangles.

Algorithm 1: Our Framework

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

```
/* Lines 1-7 are Stage-I                                              */
1  (P*, ub) ← kPlex-Degen(G, k);
2  if |P*| < ub then
3      lb ← max{|P*|, 2k − 2};
4      G ← the (lb + 1 − k)-core of G;
5      for each v ∈ V(G) do deg(v) ← the degree of v in G;
6      Compute the triangle count Δ(·,·) for all edges of G;
7      CTCP(G, ∅, true, lb + 1 − k, lb + 1 − 2k, deg, Δ);  /* core-truss
         co-pruning */;
       /* Lines 8-15 are Stage-II                                     */
8      while V(G) ≠ ∅ do
9          u ← the vertex with the minimum degree in G;
10         g ← the subgraph of G induced by vertices N_G^{≤2}(u);
11         P ← the maximum k-plex in g that contains u and is of
             size at least lb + 1 by invoking BBMatrix;   /* P ← ∅ if
             there is no such a k-plex */;
12         lb_changed ← false;
13         if |P| > lb then
14             P* ← P; lb ← |P|; lb_changed ← true;
15         CTCP(G, {u}, lb_changed, lb + 1 − k, lb + 1 − 2k, deg, Δ);
             /* remove u from G, and conduct core-truss co-pruning */;
16 return P*;
```

In addition, we also apply graph reduction techniques to prune unpromising vertices and edges from the input graph G, both before the iterative process and after removing u from the graph at Step (3).

The pseudocode of our framework is shown in Algorithm 1. We first heuristically compute a large k-plex P^* as well as an upper bound ub of the maximum k-plex size, by invoking kPlex-Degen which will be introduced shortly (Line 1). If $|P^*| = ub$, then the heuristically computed k-plex P^* is guaranteed to be maximum and the algorithm finishes. Otherwise $|P^*| < ub$ (Line 2), we use the size of P^* as the lower bound lb to reduce G to its $(lb + 1 - k)$-core (Line 4); this is because we are now searching for k-plexes of size at least $lb + 1$. Note that, if the heuristically computed k-plex is of size smaller than $2k-2$, then we set $lb = 2k-2$ as we are only interested in k-plexes of size at least $2k-1$ (Line 3). Then, we obtain the degree $\deg(\cdot)$ for vertices of G (Line 5), and compute the triangle count $\Delta(\cdot, \cdot)$ for all edges of G (Line 6), where $\Delta(u, v) = |N_G(u) \cap N_G(v)|$ is the number of triangles containing the edge (u, v). After that, we conduct core-truss co-pruning based on the computed $\deg(\cdot)$ and $\Delta(\cdot, \cdot)$ by invoking CTCP (Line 7) which will be discussed in Section 4; note that, CTCP also updates $\deg(\cdot)$ and $\Delta(\cdot, \cdot)$ when vertices and/or edges are removed from G. Then, we iteratively compute the maximum k-plex that contains u for each $u \in V(G)$ that is not pruned (Lines 9–11), by invoking BBMatrix which will be introduced in Section 5. After computing the maximum k-plex containing u, we remove u from G and then conduct a core-truss co-pruning by invoking CTCP, as other vertices and/or edges may now be able to be pruned as a result of u being removed (Line 15); here, *lb_changed* indicates whether the lower bound has changed or not since the previous invocation of CTCP. Note that, as CTCP may

remove vertices at Line 15, Lines 10–11 are not executed for every vertex of G. We will analyze the time complexity of Algorithm 1 in Theorem 4.4 after introducing the CTCP algorithm.

As we will show in Section 5 that BBMatrix represents its input graph g as an adjacency matrix, it would be beneficial to reduce the number of vertices of g that is passed to BBMatrix at Line 11 of Algorithm 1 as this then reduces the memory footprint of the adjacency matrix. Thus, in our implementation we extract a smaller subgraph than $G[N_G^{\leq 2}(u)]$ at Line 10 of Algorithm 1, without first constructing $G[N_G^{\leq 2}(u)]$. The main idea is based on Lemma 3.1 and the following lemma.

Lemma 3.2. *Let u be a vertex in a k-plex P of size $\ell \geq 2k+1$. Every vertex in the subgraph induced by u's neighbors $N_P(u)$ must have a degree at least $\ell - 2k$.*

Proof. Firstly, according to the definition of k-plex, we have $|N_P(u)| \geq \ell - k$. Secondly, as k-plex is hereditary, $N_P(u)$ is also a k-plex. Consequently, every vertex $v \in N_P(u)$ has at least $|N_P(u)| - k$ neighbors in $N_P(u)$, which is at least $\ell - 2k$. □

Thus, given a lower bound lb of the maximum k-plex size, at Line 10 of Algorithm 1 we first obtain the set of neighbors $N_G(u)$ of u and denote it by X. Following Lemma 3.2 we iteratively remove from X all vertices that have less than $lb + 1 - 2k$ neighbors in X; that is, we reduce X to its $(lb + 1 - 2k)$-core. Then, following Lemma 3.1, we obtain the set Y of non-neighbors of u that share at least $lb - 2k + 3$ common neighbors with u in X; that is, each vertex v of Y satisfies $(u, v) \notin E(G)$ and $|N_X(u) \cap N_X(v)| \geq lb - 2k + 3$. Finally, we let g be the subgraph of G induced by $X \cup Y$; note that, $X \cup Y \subseteq N_G^{\leq 2}(u)$.

Algorithm 2: kPlex-Degen$(G = (V, E), k)$

Output: A large k-plex P in G, and an upper bound ub of the maximum k-plex size in G

```
1  P ← ∅; ub ← 0;
2  for each vertex v ∈ V do deg(v) ← the degree of v in G;
3  for i ← 1 to |V| do
4      v_i ← arg min_{v ∈ V \ {v_1,...,v_{i-1}}} deg(v);
5      if deg(v_i) + k ≥ |V| − i + 1 and |V| − i + 1 > |P| then
6          P ← V \ {v_1, ..., v_{i-1}};
7      if min{deg(v_i) + k, |V| − i + 1} > ub then
8          ub ← min{deg(v_i) + k, |V| − i + 1};
9      for each v ∈ N_G(v_i) do deg(v) ← deg(v) − 1;
10 return (P, ub);
```

Heuristically Compute a Large k-Plex. Recall that Line 1 of Algorithm 1 invokes kPlex-Degen to heuristically compute a large k-plex, whose size serves as a lower bound of the maximum k-plex size and is used for pruning unpromising vertices and edges from the input graph. The pseudocode of kPlex-Degen is shown in Algorithm 2, where the k-plex is obtained by iteratively removing the vertex with the smallest degree in a similar way to the peeling algorithm for core decomposition (as discussed in Section 2.1). In Algorithm 2, we also compute an upper bound ub of the maximum k-plex size in G. The algorithm runs for $|V|$ iterations (Lines 4–9). In each iteration, it first obtains the vertex v_i that has the smallest degree in the current graph (Line 4); if there is a tie, an arbitrary vertex with the smallest degree is selected. If the current graph is a k-plex (i.e., $deg(v_i) + k \geq |V| - i + 1$) and its size is larger than P (i.e., $|V| - i + 1 > |P|$), then P is updated by the current graph (Lines 5–6). It also updates the upper bound ub (Lines 7–8), based on an upper bound of the maximum k-plex containing v_i in the current graph, which is $\min\{deg(v_i) + k, |V| - i + 1\}$. Finally, it (virtually) removes v_i and its associated edges from the graph (Line 9).

The time complexity of Algorithm 2 is $O(|V| + |E|)$. Note that, as Algorithm 2 also computes an upper bound ub of the maximum k-plex in G, the heuristically computed k-plex P is a maximum k-plex if its size is the same as ub (Line 2 of Algorithm 1).

4 EFFICIENT CORE-TRUSS CO-PRUNING

Stage-I of our framework (i.e., Lines 1–7 of Algorithm 1) removes unpromising vertices and edges from the input graph based on a heuristically computed k-plex. We call the resulting graph of G obtained after Line 7 the *kernel graph*. In this section, we propose an efficient core-truss co-pruning algorithm to compute a small kernel graph, based on the following two lemmas.

Lemma 4.1 (Core Pruning [11]). *The minimum degree of a k-plex of size ℓ is at least $\ell - k$.*

Lemma 4.2 (Truss Pruning [12]). *Any two adjacent vertices in a k-plex of size $\ell \geq 2k+1$ must have at least $\ell - 2k$ common neighbors, i.e., each edge participates in at least $\ell - 2k$ triangles, in the k-plex.*

Following Lemma 4.2, the edge between vertices u and v that have less than $\ell - 2k$ common neighbors cannot be in a k-plex of size at least ℓ.[2] Thus, we can safely remove the edge (u, v) from the graph if we are searching for a k-plex of size at least ℓ. Let lb be the size of a heuristically computed k-plex or the currently found largest k-plex. Then, we will be searching for a k-plex of size at least $lb + 1$ when computing the maximum k-plex, and we thus can reduce the graph G based on the above two lemmas as follows.

- **Vertex Reduction:** We can remove from G all vertices whose degrees are smaller than $lb + 1 - k$, i.e., *we reduce G to its $(lb + 1 - k)$-core.*
- **Edge Reduction:** We can remove from G all edges that participate in less than $lb + 1 - 2k$ triangles, i.e., *we reduce G to its $(lb + 1 - 2k)$-truss.*

Note that, a $(lb+1-k)$-core is not equivalent to a $(lb+1-2k)$-truss. That is, a $(lb+1-k)$-core is not necessarily a $(lb+1-2k)$-truss, and a $(lb+1-2k)$-truss may not necessarily be a $(lb+1-k)$-core. Thus, we will need to conduct these two reduction steps iteratively and exhaustively to reduce G as much as possible. Then, the remaining graph is the maximal subgraph that is both a $(lb + 1 - k)$-core and a $(lb + 1 - 2k)$-truss. We call this process the *core-truss co-pruning*.

Example 4.3. Consider the graph in Figure 1 and suppose $lb = 4$ and $k = 2$. First, the core pruning removes vertex v_{14} from the graph as $lb + 1 - k = 3$. Second, the truss pruning removes edge (v_1, v_4) from the graph as $lb + 1 - 2k = 1$. Third, the core pruning removes vertex v_1. Forth, the truss pruning removes edge (v_2, v_6).

[2] Together with Lemma 3.1, we can also conclude that u and v will not appear together in a k-plex of size at least ℓ, but we will not utilize this fact in the core-truss co-pruning.

Fifth, the core pruning removes all vertices except $v_{10}, v_{11}, v_{12}, v_{13}$. The resulting graph then is the kernel graph that cannot be further reduced by the current lb.

A naive algorithm to conduct the core-truss co-pruning would be iteratively applying the vertex reduction followed by edge reduction until convergence. However, the time complexity would be $O(r \cdot |E|^{1.5})$ where r is the number of iterations and can be as large as $|V|$, since each iteration of edge reduction takes time $O(|E|^{1.5})$. This time complexity is too high for large real graphs.

Algorithm 3: CTCP($G, Q_v, lb\_changed, \tau_v, \tau_e, \deg, \Delta$)

Output: Remove vertices of Q_v from G, and then reduce G to its maximal subgraph that is both a τ_v-core and a τ_e-truss

```
1  Initialize an edge queue Q_e ← ∅;
2  if lb_changed then
3      for each edge (u, v) ∈ E(G) do
4          if Δ(u, v) < τ_e then Q_e ← Q_e ∪ {(u, v)};
5  truss-peeling(G, Q_v, Q_e, τ_e, deg, Δ);
6  while there is a vertex v ∈ V(G) s.t. deg(v) < τ_v do
7      truss-peeling(G, {v}, ∅, τ_e, deg, Δ);

   Procedure truss-peeling(G, Q_v, Q_e, τ_e, deg, Δ)
8  while Q_v ≠ ∅ or Q_e ≠ ∅ do
9      while Q_e ≠ ∅ do
10         (u, v) ← pop an edge from Q_e;
11         Remove edge (u, v) from G;
12         deg(u) ← deg(u) − 1; deg(v) ← deg(v) − 1;
13         for each w ∈ N_G(u) ∩ N_G(v) do
14             Δ(u, w) ← Δ(u, w) − 1; Δ(v, w) ← Δ(v, w) − 1;
15             if Δ(u, w) + 1 = τ_e then Q_e ← Q_e ∪ {(u, w)};
16             if Δ(v, w) + 1 = τ_e then Q_e ← Q_e ∪ {(v, w)};
17     if Q_v ≠ ∅ then
18         u ← pop a vertex from Q_v;
19         for each w ∈ N_G(u) do
20             Remove edge (u, w) from G;
21             deg(w) ← deg(w) − 1;
22         for each pair of neighbours v, w ∈ N_G(u) do
23             if (v, w) ∈ E(G) then
24                 Δ(v, w) ← Δ(v, w) − 1;
25                 if Δ(v, w) + 1 = τ_e then Q_e ← Q_e ∪ {(v, w)};
26         Remove vertex u from G;
```

We propose an efficient algorithm CTCP to conduct the core-truss co-pruning with a time complexity of $O(\delta(G) \times |E|)$; recall that $\delta(G) \leq \sqrt{|E|}$ is the degeneracy of G. The pseudocode of CTCP is shown in Algorithm 3. We present the algorithm in a more general form, by taking a general core pruning threshold τ_v and a general truss pruning threshold τ_e as input. The degrees of vertices and the triangle counts of edges are, respectively, stored and maintained in deg and Δ, which are initialized at Lines 5–6 of Algorithm 1. The input boolean variable $lb\_changed$ is used to indicate whether we need to collect the set of edges whose triangle counts are smaller than τ_e; note that an invocation to CTCP (e.g., see Lines 7 and 15 of Algorithm 1) should guarantee that if $lb\_changed$ is false, then the input graph to CTCP is itself a τ_e-truss. In addition, CTCP also takes as input a set/queue of vertices Q_v that must be removed from the graph; this Q_v is obtained from Line 15 of Algorithm 1. In summary, CTCP removes vertices of Q_v from G and then reduces G to its maximal subgraph that is both a τ_v-core and a τ_e-truss.

The main idea of CTCP is to first conduct truss pruning, and then as long as there is a vertex whose degree is smaller than the core pruning threshold, we remove the vertex from the graph and conduct truss pruning again. For efficiency consideration, we collect the set of edges, whose triangle counts are smaller than the truss pruning threshold, only when necessary rather than in every iteration. Details are as follows. It first collects the set of edges whose triangle counts are smaller than τ_e into an edge queue Q_e if $lb\_changed$ is true (Lines 1–4). Secondly, it invokes truss-peeling to remove vertices of Q_v and edge of Q_e from G, and then reduce the resulting graph to its τ_e-truss (Line 5). Thirdly, as long as G is not a τ_v-core, it removes a vertex of degree smaller than τ_v from G and then reduces the resulting graph to its τ_e-truss (Lines 6–7). It is easy to verify that when the algorithm terminates, the resulting graph is the largest subgraph that is both a τ_v-core and a τ_e-truss.

The pseudocode of truss-peeling is also shown in Algorithm 3, where Q_e stores the set of edges that need to be removed as their triangle counts are smaller than τ_e. We first remove all edges of Q_e from G (Lines 9–16), and then remove a vertex $u \in Q_v$ and its associated edges from G (Lines 18–26). Note that, during each of these two processes, we also (1) update the degrees (i.e., deg) and triangle counts (i.e., Δ) for those vertices and edges that are affected (Lines 12, 14, 21, 24), and (2) push an edge into Q_e if its triangle count decreases from τ_e to $\tau_e - 1$ (Lines 15, 16, 25).

We prove the time complexity of CTCP, and more generally, the total time complexity of all invocations to CTCP by Algorithm 1, in the theorem below.

Theorem 4.4. *The time complexity of Algorithm 1, after excluding the time complexities of Lines 10–11, is $O(\delta(G) \times |E|)$ when assuming k is bounded by a small constant. Consequently, the total time complexity of all invocations to* CTCP *is $O(\delta(G) \times |E|)$.*

The proof is provided in the Appendix.

Compare with the Existing Algorithms. Although the existing algorithms BnB [11], KpLeX [12], and Maplex [38] also compute a kernel graph, CTCP differs from them in the following three aspects. Firstly, BnB and KpLeX only remove vertices to obtain the kernel graph, and the kernel graph computed by our algorithm CTCP is guaranteed to be no larger than that computed by BnB and KpLeX since "unsupported" edges, as defined in [12], are directly removed by our edge reduction. Moreover, CTCP has a lower time complexity than KpLeX for computing the kernel graph, where the time complexity of the latter is $O(r \cdot |V| \cdot |E|)$; here, r is the number of iterations which can be as large as $|V|$. Secondly, although Maplex also utilizes the idea of edge reduction to compute the kernel graph, it has a high time complexity of $O(r \cdot |E|^{1.5})$, the same as the naive algorithm discussed above. Due to the high time complexity, the implementation of Maplex open-sourced at https://github.com/ini111/Maplex does not conduct vertex reduction and edge reduction until convergence (see Table 8 in Section 6 for empirical comparisons). Thirdly, our CTCP algorithm is not only used in the preprocessing step (i.e., Stage-I) to compute the kernel

graph, but also iteratively invoked by our framework in Stage-II to reduce the graph size whenever a vertex has been processed and removed (see Line 15 of Algorithm 1).

5 A BRANCH-AND-BOUND ALGORITHM FOR DENSE GRAPHS

In State-II of our framework (i.e., Lines 8–15 of Algorithm 1), we iteratively extract a small subgraph g for computing a maximum k-plex containing a vertex u for each vertex in the graph (see Lines 10–11 of Algorithm 1). Specifically, g is a subgraph of G induced by a vertex subset of $N_G^{\leq 2}(u)$, as discussed in Section 3. Our empirical study shows that the extracted subgraphs g are dense, i.e., with high density as measured by $\frac{2|E(g)|}{|V(g)|\times(|V(g)|-1)}$; please see Table 4 in Section 6 for the numbers. In this section, we propose a branch-and-bound algorithm BBMatrix to efficiently process the extracted small and dense subgraphs g. In the following, we first present the branching, reduction and bounding techniques in Section 5.1, then give the pseudocode of BBMatrix in Section 5.2, and finally discuss our incremental computation techniques in Section 5.3.

5.1 Branching, Reduction and Bounding

In the following discussions, we represent the recursively generated instances of the branch-and-bound algorithm by $I = (g, k, S, lb)$, where $S \subseteq V(g)$ is a k-plex. For a given instance (g, k, S, lb), it aims to find the largest k-plex $P \subseteq V(g)$ such that $P \supseteq S$ and $|P| > lb$. Note that, if the lower bound lb is not relevant to the discussion, then we also represent an instance by $I = (g, k, S)$.

Branching Rules. The idea of branching rules is to select which vertex of $V(g) \setminus S$ to be processed next. That is, two branches will be generated based on the selected branching vertex u: one includes u into S, and the other excludes u from g and thus also from S. We prove in the lemma below that if there is a vertex $u \in V(g) \setminus S$ such that u has only zero or one non-neighbor in g (i.e., $d_g(u) \geq |V(g)| - 2$), then among all maximum k-plexes containing S in g, there must exist one that also contains u. Thus, we choose u as the branching vertex, and then greedily add u to S without comprising the correctness; that is, we do not generate the other branch of excluding u from S in this case. This reduces the number of newly generated branches from two to one. We term this branching rule as **BR1**; we note that this branching rule is also a reduction rule (i.e., u is in some maximum k-plex containing S).

Lemma 5.1. *Given an instance $I = (g, k, S)$,*

BR1. *For a vertex $u \in V(g) \setminus S$, if $d_g(u) \geq |V(g)| - 2$ and $S \cup \{u\}$ is a k-plex, then u is in some maximum k-plex containing S in g.*

Proof. Let's consider a maximum k-plex P in I which does not contain u, i.e., $S \subseteq P \subseteq V(g)$. Then it must be the case that there is a vertex $v \in P$ such that v is not adjacent to u and $d_P(v) = |P| - k$, as otherwise $P \cup \{u\}$ is a valid k-plex. It is easy to verify that $P \cup \{u\} \setminus \{v\}$, which contains u, is a k-plex of the same size as P. □

We remark that the above lemma cannot be directly extended to handle vertices with two or more non-neighbors in g.

If the above condition is not satisfied, then we use the branching rule that is proposed by Gao et al. [11], which we call **BR2**.

BR2. Let V_0 be the set of vertices of $V(g) \setminus S$ that have exactly $k - 1$ non-neighbors in S, i.e., $V_0 = \{v \in V(g) \setminus S \mid |S \setminus N_S(v)| = k - 1\}$. If $V_0 = \emptyset$, then V_0 is reset as $V(g) \setminus S$. Among all vertices of V_0, the branching vertex is selected as the one that has the largest degree in $V(g) \setminus S$.

That is, priority is given to vertices that have $k - 1$ non-neighbors in S. The rational is that after such a vertex u is added to S, then all non-neighbors of u in $V(g) \setminus S$ can be removed from g as only neighbors of u can be further added to S.

Reduction Rules. The general idea of reduction rules is to remove unpromising vertices from g. Given an instance $I = (g, k, S, lb)$, a vertex $v \in V(g) \setminus S$ is unpromising if either $S \cup \{v\}$ is not a k-plex, or every k-plex containing $S \cup \{v\}$ in g will be of size at most lb. We first use the following three reduction rules that have been widely used by the existing approaches, e.g., [37]

RR1. For a vertex $v \in V(g) \setminus S$, if v has at least k non-neighbors in S (i.e., $|S \setminus N_S(v)| \geq k$), then we can remove v from g as $S \cup \{v\}$ is not a k-plex.

RR2. For a vertex $v \in V(g) \setminus S$, if v has a non-neighbor $u \in S$ such that $|S \setminus N_S(u)| = k$, then we can remove v from g as $S \cup \{v\}$ is not a k-plex.

RR3. For a vertex $v \in V(g) \setminus S$, if $d_g(v) + k \leq lb$, then we can remove v from g as all k-plexes containing $S \cup \{v\}$ will be of size at most lb.

Note that **RR3** is the same as vertex reduction used in Section 4.

In addition, we propose two reduction rules, based on an upper bound of the maximum k-plex containing S in g as computed by the lemma below.

Lemma 5.2. *Given a graph g, a k-plex $S \subset V(g)$, and two vertices $u, v \in S$, the maximum size k-plex containing S in g is at most $|S| + r_S(u) + r_S(v) + \mathsf{cn}_{\bar{S}}(u, v)$. Here, $r_S(u) = k - |S \setminus N_S(u)|$ is the maximum number of non-neighbors of u outside S that can be included in any k-plex containing S, and $\mathsf{cn}_{\bar{S}}(u, v) = |N_{\bar{S}}(u) \cap N_{\bar{S}}(v)|$ is the common neighbors of u and v in $\bar{S}$ (i.e., outside S) where $\bar{S} = V(g) \setminus S$.* [3]

The proof is provided in the Appendix.

Following Lemma 5.2, we have the following two reduction rules.

RR4. For a vertex $v \in V(g) \setminus S$, if there exists a vertex $u \in S$ such that $|S| + 1 + r_{S\cup\{v\}}(u) + r_{S\cup\{v\}}(v) + \mathsf{cn}_{\bar{S}}(u, v) \leq lb$, then we can remove v from g; note that u may be adjacent to v or not.

RR5. For an edge $(u, v) \in E(g)$ where $u, v \notin S$, if $|S| + 2 + r_{S\cup\{u,v\}}(u) + r_{S\cup\{u,v\}}(v) + \mathsf{cn}_{\bar{S}}(u, v) \leq lb$, then we can remove the edge (u, v) from g.

RR4 and **RR5** are second-order reductions that consider a pair of vertices u and v. Note that, they are different from the edge reduction used in Section 4. Firstly, both **RR4** and **RR5** have another non-empty vertex set S. Secondly, in **RR4**, u and v can be either adjacent or non-adjacent, and u could be in S.

Upper Bounds. The general idea of upper bounds is to compute an upper bound of the maximum k-plex size containing S in g, such that we can prune the entire branch/instance $I = (g, k, S)$ if its

[3] Note that Lemma 5.2 generalizes the ones proved in existing studies [11, 12], by allowing S to be an arbitrary k-plex and u and v can be either adjacent or non-adjacent.

Algorithm 4: BBMatrix(g, k, lb)

Input: A (dense) graph g represented by its adjacency matrix, an integer k, and a lower bound lb of the maximum k-plex size

1 $P \leftarrow$ heuristically compute a k-plex of g;
2 **if** $|P| > lb$ **then** $P^* \leftarrow P$; $lb \leftarrow |P|$;
3 $g \leftarrow$ the $(lb + 1 - k)$-core of g;
4 **for each** $v \in V(g)$ **do** $d(v) \leftarrow$ the degree of v in g;
5 **for each** $u, v \in V(g)$ *s.t.* $u \neq v$ **do** $\mathsf{cn}(u, v) = |N_g(u) \cap N_g(v)|$;
6 CTCP$(g, \emptyset, \text{true}, lb + 1 - k, lb + 1 - 2k, d, \mathsf{cn})$;
7 BBSearch$(g, k, \emptyset, d, \mathsf{cn})$;
8 **return** P^*;

Procedure BBSearch$(g, k, S, d, \mathsf{cn})$
9 **if** $S = V(g)$ **then**
10 **if** $|S| > lb$ **then** $P^* \leftarrow S$; $lb \leftarrow |S|$;
11 **else**
12 $(u, must\_choose) \leftarrow$ choose a branching vertex from $V(g) \backslash S$;
/* The first branch includes u into S */
13 $S \leftarrow S \cup \{u\}$;
14 Reduce g based on k and S by using reduction rules;
15 **if** UB$(g, k, S, d, \mathsf{cn}) > lb$ **then** BBSearch$(g, k, S, d, \mathsf{cn})$;
16 Undo the changes made for g at Line 14;
17 $S \leftarrow S \setminus \{u\}$;
18 **if** *must_choose* = false **then**
/* The second branch excludes u from g */
19 Remove u from g;
20 Reduce g based on k and S by using reduction rules;
21 **if** UB$(g, k, S, d, \mathsf{cn}) > lb$ **then** BBSearch$(g, k, S, d, \mathsf{cn})$;
22 Undo the changes made for g at Line 20;
23 Add u back to g;

upper bound is no larger than lb. In this paper, we use two upper bounds, and take the minimum among them.

UB1. $\min_{u \in S}\{d_g(u)\} + k$.

UB2. $\min_{u,v \in S, u \neq v}\{r_S(u) + r_S(v) + \mathsf{cn}_{\bar{S}}(u, v)\} + |S|$.

UB1 has been widely used in the literature. The correctness of **UB2** directly follows from Lemma 5.2. Note that, in computing **UB2**, u and v can be either adjacent or non-adjacent.

Note that, reduction rules **RR3**, **RR4**, **RR5** share similar ideas with the upper bounds, i.e., these three reduction rules are also based on upper bounds. But they are different in the following ways. Firstly, **RR3** computes upper bounds for vertices of $V(g) \setminus S$ and prunes vertices from g based on the computed upper bounds, while **UB1** computes upper bounds for vertices of S and prunes the entire branch/instance based on the computed upper bounds. Secondly, **RR4** computes upper bounds for pairs of vertices where one is from S and the other is from $V(g) \setminus S$, while **UB2** computes upper bounds for pairs of vertices that are both from S. Lastly, **RR5** computes upper bounds for pairs of vertices that are both from $V(g) \setminus S$ and prunes edges based on the computed upper bounds.

5.2 Pseudocode of BBMatrix

Based on the branching, reduction, and upper bounding techniques discussed in Section 5.1, the pseudocode of BBMatrix is shown in Algorithm 4. Note that we store the input graph g to BBMatrix as an adjacency matrix, because (1) g is small and dense and (2) we will need to compute and store $\mathsf{cn}(\cdot, \cdot)$ for all pairs of vertices of g that occupies a quadratic space w.r.t. $|V(g)|$. In our implementation, we map the vertex ids of $V(g)$ into consecutive integers in $\{0, 1, \ldots, |V(g)| - 1\}$, such that the adjacency matrix of g is stored in an array of size $|V(g)| \times |V(g)|$.

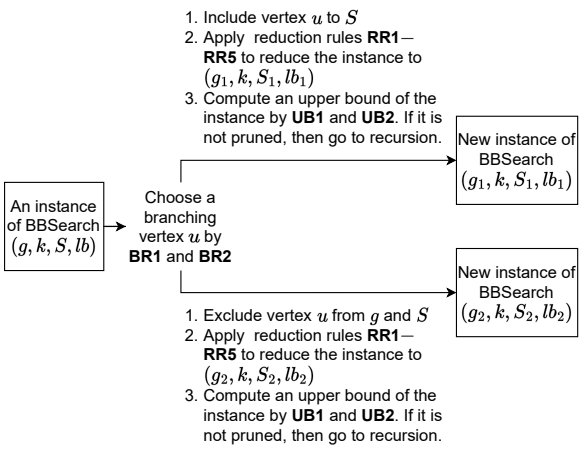


Figure 2: An overview of the flow of the BBSearch algorithm

BBMatrix first preprocesses g by heuristically computing a k-plex (Line 1), reducing g to its $(lb+1-k)$-core (Line 3), and conducting core-truss co-pruning (Line 6), in a similar way to Algorithm 1. However, instead of only computing the triangle counts for all edges of g, we compute the number of common neighbors for each pair of vertices (whether they are adjacent or not), denoted $\mathsf{cn}(\cdot, \cdot)$, on Line 5. This is because we will later use $\mathsf{cn}(u, v)$ for reduction, pruning, and upper bounding no matter u and v are adjacent or not. After that, we call our branch-and-bound search algorithm BBSearch for computing a maximum k-plex in g.

The pseudocode of BBSearch is also given in Algorithm 4; an overview of the flow of BBSearch is shown in Figure 2. We represent a branch-and-bound instance by $I = (g, k, S, lb)$, where $S \subseteq V(g)$ is a k-plex in g, and we aim to compute a maximum k-plex of size larger than lb that contains S in g. Note that, $d(\cdot)$ and $\mathsf{cn}(\cdot, \cdot)$ maintained by the algorithm actually represent $d_g(\cdot)$ and $\mathsf{cn}_{\bar{S}}(\cdot, \cdot)$; we omit the subscript for notation simplicity. We first select a branching vertex from $V(g) \setminus S$, denoted by u, by branching rules **BR1** and **BR2** (Line 12). Note that, if the branching rule **BR1** is actually in effect here, then u can be greedily added to S without backtracking and *must_choose* will be true; otherwise, *must_choose* will be false. Then we generate the first branch, which includes u into S (Lines 13–15). After the branch, we undo the changes that we have made in this branch (Lines 16–17). This is because we may update g (i.e., remove edges from g), d and cn throughout the algorithm; recall that, for an instance $I = (g, k, S, lb)$, $\mathsf{cn}(u, v)$ stores the number of common neighbors of u and v in $V(g) \setminus S$. If u is not determined to be must included (Line 18, see **BR1**), we similarly generate the second branch that excludes u from g (Lines 19–21). Note that, after generating each branch, we first apply reduction rules, **RR1** – **RR5**, to reduce the size of the instance (Lines 14 and 20), and then prune the instance if its upper bound computed by either **UB1** or **UB2** is no larger than lb (Lines 15 and 21). Note that at the point that

reduction rules are applied or UB is evaluated, a vertex u's degree $d(u)$ is the degree of u in g (i.e., $d_g(u)$) and the number of common neighbors $\mathsf{cn}(u, v)$ of vertices u and v is the common neighbors in $\bar{S}$ (i.e., $\mathsf{cn}_{\bar{S}}(u, v)$).

Compared to the existing branch-and-bound algorithms for maximum k-plex computation, our algorithm BBMatrix has two unique features. Firstly, our algorithm incorporates second-order techniques for reduction (i.e., **RR4** and **RR5**) and pruning (i.e., **UB2**) in the recursion, while the existing algorithms only use first-order information in the recursion. Secondly, we apply the reduction and bounding techniques efficiently, based on incremental computation techniques as introduced next.

5.3 Incremental Computation

For efficiency, we apply the reduction rules and upper bound-based pruning incrementally, instead of blindly trying all the reduction rules and all the upper bounds. That is, we only check the vertices or the vertex pairs that may be pruned due to the updating of the instance I. This is achieved by maintaining d and cn. For example, whenever we update $\mathsf{cn}(u, v)$ for vertices u and v, we apply one of **RR4**, **RR5**, and **UB2** depending on how many of u and v are in S.

- If both u and v are in S, then we check whether **UB2** can be applied to prune the instance.
- If both u and v are not in S and there is an edge between u and v, then we check whether **RR5** can be applied to remove the edge (u, v) from g.
- If only one of u and v is in S, assume $u \in S$, then we check whether **RR4** can be applied to remove v from g.

As $\mathsf{cn}_{\bar{S}}(u, v)$, $d_g(u)$, and $d_g(v)$ are all maintained (i.e., in $\mathsf{cn}(\cdot, \cdot)$ and $d(\cdot)$), each of the three checks can be conducted in constant time. In this way, our reduction rules and upper bounds are applied almost without extra cost when maintaining $d(\cdot)$ and $\mathsf{cn}(\cdot, \cdot)$.

Further details are as follows. For every vertex $v \in V(g)$, besides maintaining the degree $d(v)$ of v in g, we also maintain the number of neighbors of v in S, denoted $d_S(v)$. Thus, $r_{S\cup\{v\}}(v) = k - (|S \cup \{v\}| - d_S(v))$; here, v can be either inside S or outside S. For notation simplicity, we let $\mathsf{UB}(v, w) = |S \cup \{v, w\}| + r_{S\cup\{v,w\}}(v) + r_{S\cup\{v,w\}}(w) + \mathsf{cn}_{\bar{S}}(v, w)$ be the upper bound computed by Lemma 5.2. In the first branch that includes u into S (i.e., Lines 13–15), we first add u to S and thus $|S|$ increases by 1, and then conduct the following operations.

(1) For each neighbor v of u, $d_S(v)$ increases by 1.
(2) For each non-neighbor v of u,
   (a) If $v \notin S$ and $|S| - d_S(v) \geq k$, remove v from g by reduction rule **RR1**; note that although $d_S(v)$ does not change, $|S|$ has increased by 1.
   (b) If $v \in S$ and $|S| - d_S(v) = k$, remove from g all of v's non-neighbors in $V(g) \setminus S$ by reduction rule **RR2**.
(3) For each non-neighbor $v \in V(g) \setminus S$ of u, check whether v can be removed by reduction rule **RR4**; note that both $\mathsf{cn}(u, v)$ and $\mathsf{UB}(u, v)$ remain unchanged for each vertex $v \in V(g) \setminus \{u\}$.
(4) For each pair of neighbors v, w of u, $\mathsf{cn}(v, w)$ decreases by 1 while $\mathsf{UB}(v, w)$ does not change.
(5) For each pair of non-neighbors v, w of u, $\mathsf{cn}(v, w)$ does not change, but $\mathsf{UB}(v, w)$ decreases by 1. If $\mathsf{UB}(v, w) \leq lb$, then
   (a) If $v, w \in S$, prune the entire branch by **UB2**.
   (b) If $v, w \in V(g) \setminus S$ and $(v, w) \in E(g)$, remove the edge from g by reduction rule **RR5**.
   (c) If $v \in S, w \in V(g) \setminus S$, remove w from g by reduction rule **RR4**.

Note that, once a vertex or an edge is removed during the above operations, we also need to update $d_S(\cdot), d(\cdot)$ and $\mathsf{cn}(\cdot, \cdot)$ for their neighbors and then iteratively apply the reductions and upper bound-based pruning, in a similar way to CTCP (Algorithm 3); we omit the details of the cascading reduction. For the time complexity analysis, if we do not consider the cascading reduction, then it is $O(|V(g)|^2)$; that is, items (1)–(3) take $O(|V(g)|)$ time in total and items (4)–(5) take $O(|V(g)|^2)$ time in total. If we consider the cascading reduction, then the time complexity of BBSearch without going into the recursion is $O(|V(g)|^3)$ since in the worst case $|V(g) \setminus S|$ vertices and $|V(g) \setminus S|^2$ edges could be removed during the cascading reduction.

For the second branch that excludes u from g, we conduct similar operations; details are omitted.

6 EXPERIMENTS

We evaluate the efficiency of our techniques for maximum k-plex computation, by comparing the following algorithms.

- BnB [4]: the existing algorithm proposed in [11].
- Maplex [5]: the existing algorithm proposed in [38].
- KpLeX [6]: the existing algorithm proposed in [12].
- BnB-ct: our variant of BnB that replaces its preprocessing algorithm with our core-truss co-pruning algorithm CTCP.
- kPlexS: our algorithm presented in Algorithm 1.
- kPlexF: a variant of kPlexS that does not apply second-order techniques (i.e., **RR4**, **RR5**, **UB2**) in BBMatrix.

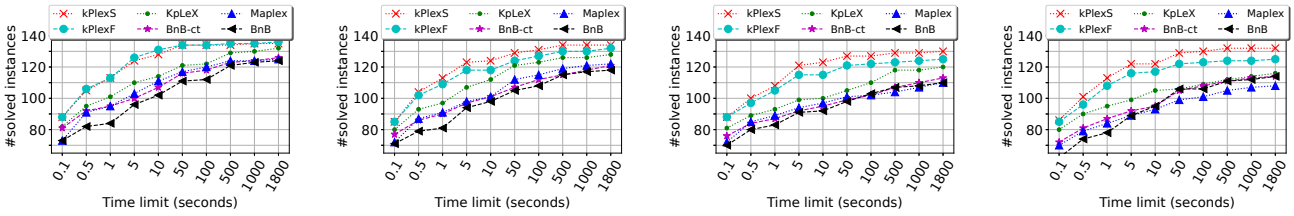
All the algorithms are implemented in C++. We conduct the experiments on a machine with an Intel(R) Xeon(R) W-2123 CPU @ 3.60GHz and 128GB main memory running Ubuntu. All the experiments are run in main memory and in single-thread mode.

Datasets. We evaluate the algorithms on two collections of graphs that have been widely used in the literature [11, 12, 38].

- **Real-world Graphs.** This collection of graphs is downloaded from http://lcs.ios.ac.cn/~caisw/Resource/realworld%20graphs.tar.gz, which contains 139 real-world graphs with up to 5.87×10^7 vertices from the Network Data Repository.
- **10th DIMACS Graphs.** This collection of graphs is downloaded from https://networkrepository.com/dimacs10.php, which contains 84 graphs with up to 5.09×10^7 vertices.

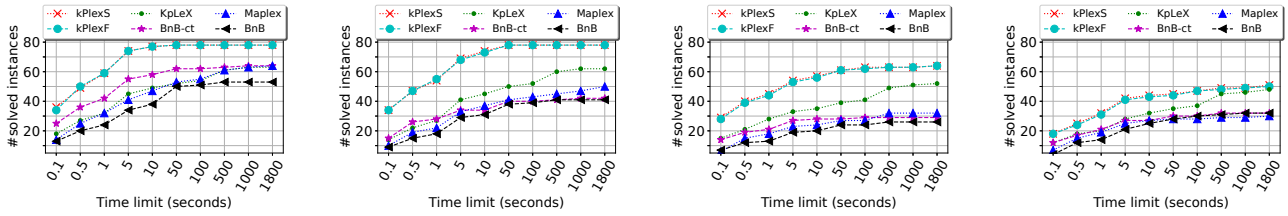
We use these graphs to conduct two sets of experiments. Firstly, we conduct a macro experiment by reporting the number of graph instances that are successfully solved by an algorithm within a specified time limit. Secondly, we conduct a micro experiment by reporting more detailed information of running these algorithms on a subset of real-world graphs. Specifically, the set of graph instances from the real-world graphs collection that either kPlexS or KpLeX

[4] https://github.com/JimNenu/codekplex
[5] https://github.com/ini111/Maplex
[6] https://github.com/huajiang-ynu/kplex



(a) $k = 2$ (b) $k = 3$ (c) $k = 5$ (d) $k = 7$

Figure 3: Number of solved instances for real-world graphs (vary time limit, best viewed in color)



(a) $k = 2$ (b) $k = 3$ (c) $k = 5$ (d) $k = 7$

Figure 4: Number of solved instances for DIMACS10 graphs (vary time limit, best viewed in color)

Table 2: Statistics of the 22 real-world graphs that either kPlexS or KpLeX takes between 10 and 1800 seconds for $k = 5$ (density= $\frac{2|E|}{|V|(|V|-1)}$, last column is the maximum 5-plex size)

ID	Graph	$\|V\|$	$\|E\|$	density	$\delta(G)$	$k = 5$
G_1	socfb-MIT	6402	251230	0.0123	72	48
G_2	scc_reality	6809	4714485	0.2034	1235	1237
G_3	tech-WHOIS	7476	56943	0.0020	88	76
G_4	socfb-Berkeley13	22900	852419	0.0033	64	53
G_5	socfb-Texas84	36364	1590651	0.0024	81	68
G_6	sc-nasasrb	54870	1311227	0.0009	35	24
G_7	soc-slashdot	70068	358647	0.0001	53	40
G_8	rec-amazon	91813	125704	0.0000	4	8
G_9	ia-wiki-Talk	92117	360767	0.0001	58	25
G_{10}	sc-pkustk13	94893	3260967	0.0007	41	36
G_{11}	soc-gowalla	196591	950327	0.0000	51	32
G_{12}	sc-pwtk	217891	5653221	0.0002	35	26
G_{13}	sc-msdoor	404785	9378650	0.0001	34	23
G_{14}	soc-youtube	495957	1936748	0.0000	49	26
G_{15}	soc-youtube-snap	1134890	2987624	0.0000	51	26
G_{16}	soc-lastfm	1191805	4519330	0.0000	70	27
G_{17}	soc-pokec	1632803	22301964	0.0000	47	34
G_{18}	web-wikipedia2009	1864433	4507315	0.0000	66	32
G_{19}	soc-flixster	2523386	7918801	0.0000	68	49
G_{20}	socfb-B-anon	2937612	20959854	0.0000	63	35
G_{21}	socfb-A-anon	3097165	23667394	0.0000	74	37
G_{22}	socfb-uci-uni	58790782	92208195	0.0000	16	13

takes at least 10s and at most 1800s to process for $k = 5$ are chosen. There are 22 such graphs. We name them as $\{G_1, G_2, \ldots, G_{22}\}$. Statistics of these 22 graphs are given in Table 2.

Measures. We report both the *processing time* and *peak memory consumption* of running an algorithm on a graph instance, where a timeout of 1800s is set. The reported processing time is the total CPU time excluding only the I/O time of loading the graph instance from disk to main memory. The peak memory usage of a program is recorded by /usr/bin/time [7]. Note that, to find the exact maximum k-plex no matter how small it is, we invoke KpLeX on the graph instance again if our algorithm kPlexS (resp. kPlexF) fails to report a k-plex of size at least $2k - 2$. *If this happens, then the reported processing time of* kPlexS *(resp.* kPlexF*) also includes that of* KpLeX.

We select k from $\{2, 3, 5, 7\}$. Note that we do not test $k = 1$, as 1-plexes are cliques which have dedicated and thus more efficient algorithms to compute (e.g., see [6]).

6.1 Experimental Results.

Number of Solved Instances for a Collection of Graphs. We first conduct a macro experiment by considering all graph instances in a graph collection, and report the number of instances that are solved by an algorithm within a specific time limit. The results of running all the six algorithms kPlexS, kPlexF, KpLeX, BnB-ct, Maplex, BnB on the real-world graph collection for $k = 2, 3, 5, 7$ are shown in Figure 3. We can see that for any given time limit, our algorithms kPlexS and kPlexF always solve the largest number of instances, and kPlexS with a time limit of 50s solves even more instances than the existing algorithms KpLeX, Maplex, BnB with a time limit of 1800s. As kPlexF only applies simple first-order techniques in the branch-and-bound search, the superiority of kPlexF over the existing algorithms is mainly due to our framework of conducting branch-and-bound search on dense subgraphs that are iteratively extracted from the input graph. Furthermore, BnB-ct, our improved version of BnB, performs better than BnB and performs similarly to Maplex. The improvement of BnB-ct over BnB is solely due to our efficient preprocessing algorithm CTCP. However, without our branch-and-bound-algorithm BBMatrix, BnB-ct is still slow. Lastly, kPlexS performs better than its variant kPlexF, especially for $k \geq 3$; note that the only different between these two algorithms is that kPlexF does not apply second-order techniques in the branch-and-bound search. This demonstrates the superiority of applying second-order techniques in the branch-and-bound search.

The results on the 10th DIMACS graph collection are reported in Figure 4. The trends are similar to Figure 3. But now we see a large drop in the number of instances solved by each algorithm

[7] https://man7.org/linux/man-pages/man1/time.1.html

Table 3: Total processing time and peak memory usage for $k = 5$ (second column shows that our algorithm kPlexS is x times faster than the best among KpLeX, Maplex, BnB-ct and BnB)

ID	Speed up	Time (seconds)					Memory (MB)				
		kPlexS	KpLeX	Maplex	BnB-ct	BnB	kPlexS	KpLeX	Maplex	BnB-ct	BnB
G_1	2.3	**0.24**	15.51	395.50	0.56	1.56	**11**	313	16	**11**	20
G_2	6.9	**8.26**	57.04	456.81	-	-	**72**	356	87	-	296
G_3	34.8	**3.95**	137.49	-	509.11	559.46	**4**	309	-	**4**	7
G_4	10.3	**0.66**	21.60	533.76	6.85	15.99	**26**	324	47	**26**	57
G_5	109.2	**1.14**	233.18	-	124.82	218.58	**36**	332	-	**36**	104
G_6	36.9	**2.62**	96.71	1490.74	-	-	**47**	339	1205	-	91
G_7	23.8	**2.20**	121.66	-	52.35	90.87	**7**	312	-	**7**	31
G_8	3088.1	**0.03**	86.48	-	-	-	**10**	314	-	-	21
G_9	19.5	**5.74**	111.73	-	947.10	1702.26	**12**	315	-	13	34
G_{10}	392.0	**1.18**	464.19	-	-	-	**110**	385	-	-	212
G_{11}	414.9	**0.16**	64.55	64.41	65.71	75.54	**17**	323	23	18	84
G_{12}	692.4	**2.58**	1789.24	-	-	-	**195**	444	-	568	377
G_{13}	-	**1761.51**	-	-	-	-	**321**	-	60599	-	-
G_{14}	21.4	**0.42**	24.14	817.83	9.01	16.81	**41**	343	61	**41**	177
G_{15}	43.3	**0.53**	62.94	1039.90	22.82	38.71	**61**	367	83	**61**	292
G_{16}	1.1	**182.59**	196.36	-	930.37	1516.73	**80**	385	-	**80**	407
G_{17}	1.0	**17.26**	18.04	37.75	33.70	72.30	**553**	707	1113	**553**	1557
G_{18}	3.8	**0.53**	190.33	2.05	-	-	**98**	390	124	-	466
G_{19}	-	**419.12**	-	-	-	-	**141**	-	-	-	-
G_{20}	7.7	**29.97**	1326.42	-	230.27	-	**628**	764	-	679	1623
G_{21}	1.8	**21.75**	255.35	39.30	52.42	178.95	**564**	748	1082	565	1779
G_{22}	0.8	15.24	**12.91**	25.94	16.33	21.68	2781	**2373**	3453	2781	11667

Table 4: Density of subgraphs (density $\frac{2|E|}{|V|\cdot(|V|-1)}$ for kernel graph (den_K) and graphs input to BBMatrix)

ID	den_K	den_{avg}	den_{min}
G_1	0.2679	0.95	0.92
G_2	1.0000	1.00	1.00
G_3	0.8878	0.94	0.90
G_4	0.2220	0.96	0.95
G_5	0.8392	0.95	0.93
G_6	0.0009	0.77	0.66
G_7	0.6836	0.84	0.77
G_8	0.0001	0.42	0.33
G_9	0.1543	0.55	0.44
G_{10}	0.0011	0.92	0.82
G_{11}	0.1331	0.85	0.78
G_{12}	0.0002	0.76	0.61
G_{13}	0.0001	0.84	0.55
G_{14}	0.1017	0.69	0.54
G_{15}	0.0980	0.72	0.60
G_{16}	0.1361	0.53	0.36
G_{17}	0.0517	0.90	0.84
G_{18}	0.0071	0.88	0.75
G_{19}	0.3557	0.83	0.75
G_{20}	0.0015	0.82	0.64
G_{21}	0.0719	0.89	0.79
G_{22}	1.0000	1.00	1.00

Table 5: Total processing time for $k = 2$

ID	kPlexS	KpLeX	Maplex	BnB-ct	BnB
G_1	**0.27**	8.25	6.13	28.32	30.64
G_2	0.40	**0.02**	0.11	0.40	0.30
G_3	**0.51**	164.37	-	-	-
G_4	**0.74**	6.03	2.02	32.99	19.55
G_5	**2.02**	188.95	-	-	-
G_6	**0.31**	6.41	22.18	441.03	490.55
G_7	**0.35**	1.00	9.34	12.79	17.09
G_8	**0.01**	0.02	0.02	0.02	0.02
G_9	2.66	**0.77**	3.01	9.96	10.24
G_{10}	**1.16**	446.31	88.94	132.77	320.22
G_{11}	**0.18**	1.05	0.60	7.73	12.42
G_{12}	**1.24**	1570.30	-	-	-
G_{13}	**8.21**	-	-	-	-
G_{14}	**0.43**	0.73	1.78	1.39	8.45
G_{15}	**0.60**	1.08	2.55	3.35	14.88
G_{16}	23.17	**4.76**	5.32	49.29	109.84
G_{17}	23.01	**12.11**	33.38	30.09	82.01
G_{18}	**0.42**	4.45	2.71	63.75	121.48
G_{19}	**2.67**	18.82	101.21	368.15	333.11
G_{20}	**25.51**	92.96	67.98	67.53	738.56
G_{21}	23.72	**19.08**	36.36	39.60	148.67
G_{22}	**8.74**	12.88	25.13	9.43	31.58

Table 6: Total processing time for $k = 3$

kPlexS	KpLeX	Maplex	BnB-ct	BnB
0.26	11.10	50.31	6.66	17.13
0.40	**0.02**	0.12	0.40	0.32
0.64	25.63	-	531.26	816.95
0.70	0.95	24.28	167.55	227.34
4.41	66.28	-	-	-
0.31	16.78	98.24	-	-
3.04	9.91	423.68	97.95	121.00
0.01	2.13	0.04	28.55	596.19
5.62	**1.91**	47.56	51.81	43.10
1.17	447.37	301.06	-	-
0.15	1.26	0.67	8.82	13.97
1.55	1582.88	-	-	-
31.75	-	-	-	-
0.48	1.25	10.11	3.58	12.07
0.78	2.38	30.41	9.07	27.88
76.62	**7.54**	37.51	175.50	314.57
20.52	**14.52**	28.57	24.65	100.28
0.41	7.49	2.53	1688.30	-
151.51	275.44	-	-	-
26.59	75.48	311.70	74.68	-
22.43	32.81	38.54	64.24	231.30
12.41	**11.67**	24.61	12.65	24.18

Table 7: Total processing time for $k = 7$

kPlexS	KpLeX	Maplex	BnB-ct	BnB
0.23	5.77	1621.59	0.23	1.34
0.40	**0.02**	0.11	0.39	0.30
0.63	6.04	-	5.83	7.08
0.62	9.03	73.92	21.79	21.57
0.84	**0.38**	7.34	0.88	6.89
0.29	94.24	976.85	-	-
3.78	624.12	-	19.76	30.59
0.01	0.02	0.02	0.01	0.01
1.74	-	-	-	-
110.69	-	-	-	-
0.14	566.02	202.47	52.78	39.32
1.12	1765.75	-	-	-
15.29	-	-	-	-
0.38	376.79	-	40.55	46.28
0.49	1804.93	-	91.87	132.67
72.80	-	-	-	-
17.26	109.23	87.52	54.56	101.20
0.69	-	4.12	-	-
-	-	-	-	-
23.03	-	-	41.02	158.42
19.04	-	40.49	176.30	476.70
15.05	**11.62**	24.04	15.04	20.17

when k increases. This is because this graph collection contains several large instances that have a very small maximum k-plex (i.e., of size smaller than $2k - 2$) for $k \geq 5$; these are the hard instances. Note that, for these instances, our reported processing time of kPlexS (resp. kPlexF) also includes that of KpLeX. If we consider only the processing time of kPlexS, then the plots for kPlexS for $k = 5, 7$ would be similar to that of $k = 2$; specifically, kPlexS finishes within the time limit for 78 instances for $k = 5, 7$. We remark that kPlexS and kPlexF perform similarly on this graph collection; this is because the benefit brought by maintaining $\mathsf{cn}(\cdot, \cdot)$ for graphs in this collection does not outweigh its overhead.

Total Processing Time and Peak Memory Usage. The total processing time and peak memory usage of running the five algorithms kPlexS, KpLeX, BnB-ct, Maplex, BnB on the 22 real graphs $G_1, \ldots, G_{22}$ for $k = 5$ are reported in Table 3. Note that, these 22 real graphs are the ones that either kPlexS or KpLeX finishes in time between 10s and 1800s; there are 20 such graphs for KpLeX and 7 for kPlexS, where the specific graphs can be identified from Table 3. We can see that our algorithm kPlexS solves 15 of these graphs within 10s, while the overall second-best algorithm KpLeX needs at least 10s for each of the 22 graphs. The second column shows the speed up of our algorithm kPlexS over the best among KpLeX, Maplex, BnB-ct and BnB; the larger the better. We can see that the speed up of kPlexS over all existing algorithms can be more than two orders of magnitude, e.g., see G_5, G_8, G_{10}, G_{11}, G_{12}. The total processing time for other k values on these graphs are reported in Tables 5, 6 and 7. The trends are similar to Table 3.

Regarding peak memory usage, we can see that our algorithm kPlexS always have the smallest memory footprint (except on

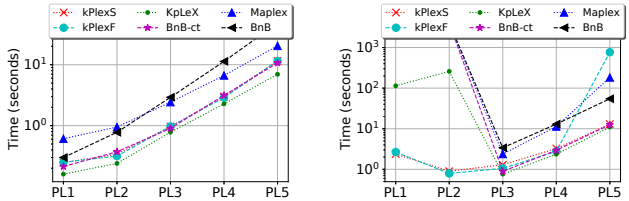
Table 8: Preprocessing for $k = 5$ (time in seconds, P is the heuristically computed k-plex, (V_K, E_K) is the kernel graph)

ID	kPlexS				KpLeX				Maplex				BnB			
	Time	$\|P\|$	$\|V_K\|$	$\|E_K\|$	Time	$\|P\|$	$\|V_K\|$	$\|E_K\|$	Time	$\|P\|$	$\|V_K\|$	$\|E_K\|$	Time	$\|P\|$	$\|V_K\|$	$\|E_K\|$
G_1	0.23	43	209	5824	0.15	43	209	5824	0.38	44	151	4183	1.24	47	60	1634
G_2	1.20	1236	1239	766923	2.42	1236	1239	766923	1.89	1236	1239	766923	349.73	1236	1239	766923
G_3	0.01	75	120	6339	0.01	75	128	6976	0.04	75	128	6976	0.10	75	121	6434
G_4	0.66	50	291	9368	0.30	50	292	9421	0.97	50	292	9421	4.88	50	292	9771
G_5	1.02	67	114	5405	0.43	68	111	5167	1.32	67	118	5686	8.03	68	111	5256
G_6	0.12	24	51153	1205204	0.54	24	51153	1205309	1.32	24	51153	1205309	3.31	24	51154	1231573
G_7	0.02	40	102	3521	0.02	39	111	3944	0.05	40	102	3521	0.10	39	113	4131
G_8	0.02	6	61351	95242	0.01	5	91813	125704	0.03	6	61351	95242	0.04	5	91813	125704
G_9	0.09	23	527	21384	0.21	21	705	29568	0.56	24	413	16478	1.77	21	715	34612
G_{10}	0.64	36	60849	1982268	1.01	27	94795	3255707	24.82	36	60849	1982268	19.37	27	94795	3258857
G_{11}	0.13	28	452	13564	0.17	27	532	16438	0.38	29	370	10738	1.80	28	460	15327
G_{12}	0.36	24	212326	5507216	4.84	24	212326	5507216	5.66	24	212326	5507216	12.43	24	212461	5524384
G_{13}	0.71	21	404785	9371658	-	-	-	-	1.51	21	404785	9371658	-	-	-	-
G_{14}	0.36	22	660	22114	0.46	22	663	22187	1.37	22	663	22187	4.80	22	700	27964
G_{15}	0.48	23	673	22154	0.61	22	984	33637	1.76	23	702	23009	9.89	22	1015	41671
G_{16}	0.82	25	660	29604	0.79	25	661	29633	1.75	25	661	29633	10.15	25	677	35674
G_{17}	17.23	27	814	17119	9.14	27	825	17403	27.01	27	825	17403	55.07	30	298	6207
G_{18}	0.50	14	4818	82808	0.88	14	4820	82848	2.00	15	3586	67144	3.32	14	4867	88046
G_{19}	0.36	46	244	10546	-	-	-	-	0.09	48	226	9407	-	-	-	-
G_{20}	23.96	21	33082	845687	30.80	21	33103	846392	39.29	23	17124	452253	240.25	21	34767	1318592
G_{21}	21.72	33	775	21559	11.86	33	777	21635	30.90	33	777	21635	87.22	33	819	27622
G_{22}	15.23	13	0	0	12.90	12	578	4556	25.94	13	0	0	21.68	13	0	0

G_{22}), despite of using the adjacency matrix graph representation in branch-and-bound search BBMatrix. This can be explained by the density (i.e., $\frac{2|E|}{|V|\cdot(|V|-1)}$) of the graphs that are input to BBMatrix. Specifically, the third and the fourth columns of Table 4, respectively, report the average density and the minimum density among all subgraphs that are solved by BBMatrix. We can see that the average densities are at least 0.42, and the minimum densities are always above 0.33. The second column of Table 4 shows the density of the kernel graph (i.e., the graph obtained after Line 7 of Algorithm 1). We can see that this is always no larger than den_{min}, and sometimes can be much smaller than den_{min}. This demonstrates the advantages of our framework which iteratively extracts dense subgraph to solve. Note that, the space complexity of BBMatrix is $O(|V(g)|^2)$, as there is only one copy of the adjacency matrix stored in the main memory; the adjacency matrix of a subgraph of g is simply a submatrix of the adjacency matrix of g.

Preprocessing Algorithms. In this testing, we evaluate the different preprocessing algorithms that are used in kPlexS, KpLeX, Maplex, and BnB for computing the kernel graph. The running time, kernel graph size (i.e., $|V_K|$ and $|E_K|$), and the heuristically computed k-plex size $|P|$ are reported in Table 8. We can see that our preprocessing algorithm CTCP that is used in kPlexS is almost always faster than existing preprocessing algorithms; this conforms with our time complexity analysis. When comparing the kernel graph sizes, we should be careful to only compare the kernel graph sizes for the different algorithms that have the same $|P|$ value; otherwise, they are incomparable. We observe that Maplex always has the same kernel graph size as KpLeX when the heuristically computed k-plexes (i.e., $|P|$) are of the same size. Consistently, our algorithm kPlexS (specifically, CTCP) always computes the smallest kernel graph when $|P|$ values are equal.

Varying Graph Density. In this testing, we evaluate the algorithms on synthetic power-law graphs that are generated by GTGraph [8], by varying the graph density. Specifically, we generate five power-law graphs, PL1, ..., PL5, with $\approx 1.3 \times 10^5$ vertices; the average degree varies from 8 to 120 with an increasing factor of ≈ 2. The results of running the six algorithms for $k = 2$ and $k = 5$ are shown in Figure 5. We can see that for $k = 2$, the running time of all the algorithms increases when the graph density increases. However, for $k = 5$, the fastest running time is achieved in the middle (i.e., on PL3); a possible reason is that the maximum k-plex sizes of PL1 and PL2 are small (i.e., 8 and 9, respectively) compared to $2k - 1$, which makes the pruning techniques less effective and thus the search space larger. Nevertheless, our algorithm kPlexS always outperform the existing algorithms, except KpLeX on some instances.



(a) $k = 2$ (b) $k = 5$

Figure 5: Running time of the algorithms on synthetic power-law graphs by varying density (best viewed in color)

7 CONCLUSION

In this paper, we proposed an efficient algorithm kPlexS for maximum k-plex computation over large sparse graphs. It incorporate three novel ingredients: a new framework, a theoretically faster and better preprocessing algorithm CTCP, and a matrix-based branch-and-bound algorithm BBMatrix that incorporates both first-order and second-order pruning techniques. Extensive experimental results on large real graphs demonstrated the efficiency of our algorithms. One possible direction of future work is to incorporate the vertex partitioning-based upper bound that is proposed in [12] and is orthogonal to our techniques into our implementation.

[8] http://www.cse.psu.edu/~madduri/software/GTgraph/

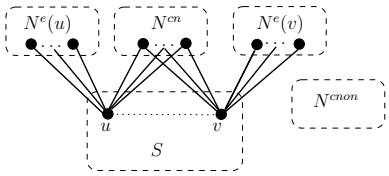


Figure 6: Proof of Lemma 5.2

REFERENCES

[1] B Balasundaram. 2007. *Graph Theoretic Generalisations of Clique: Optimisation and Extensions.* Ph.D. Dissertation. Texas A&M University, Berlin.

[2] 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.

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

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

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

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

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

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

[9] 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.

[10] 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).

[11] 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.*

[12] 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.

[13] V E Krebs. 2002. Mapping Networks of Terrorist Cells. *International Network For Social Network Analysis* 24 (2002), 43–52. Issue 3.

[14] 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.

[15] Chu-Min Li, Zhiwen Fang, and Ke Xu. 2013. Combining MaxSAT Reasoning and Incremental Upper Bound for the Maximum Clique Problem. In *Proc. of ICTAI'13.*

[16] 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.

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

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

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

[20] 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.

[21] Mark Ortmann and Ulrik Brandes. 2014. Triangle Listing Algorithms: Back from the Diversion. In *Proc. of ALENEX'14.* 1–8.

[22] Patric R. J. Östergård. 2002. A Fast Algorithm for the Maximum Clique Problem. *Discrete Appl. Math.* 120, 1–3 (2002), 197–207.

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

[24] Bharath Pattabiraman, Md. Mostofa Ali Patwary, Assefaw Hadish Gebremedhin, Wei-keng Liao, and Alok N. Choudhary. 2015. Fast Algorithms for the Maximum Clique Problem on Massive Graphs with Applications to Overlapping Community Detection. *Internet Mathematics* 11, 4-5 (2015), 421–448.

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

[26] Ryan A. Rossi, David F. Gleich, and Assefaw Hadish Gebremedhin. 2015. Parallel Maximum Clique Algorithms with Applications to Network Analysis. *SIAM J. Scientific Computing* 37, 5 (2015).

[27] 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.

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

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

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

[31] Etsuji Tomita, Yoichi Sutani, Takanori Higashi, Shinya Takahashi, and Mitsuo Wakatsuki. 2010. A simple and faster branch-and-bound algorithm for finding a maximum clique. In *Proc. of WALCOM'10.* 191–203.

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

[33] 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.

[34] 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.

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

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

[37] 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.*

[38] 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.

A APPENDIX

A.1 Proof of Theorem 4.4

It is easy to see that Lines 1–5 of Algorithm 1 run in $O(|E|)$ time. For Line 6 of Algorithm 1, the triangle counts for all edges are computed by running the K3 algorithm on the oriented graph of G which is obtained by the degeneracy ordering [21]. Thus, the time complexity of Line 6 of Algorithm 1 is $O(\delta(G) \times |E|)$, and what remains to be proved is that all invocations to CTCP also finish in $O(\delta(G) \times |E|)$ time.

Firstly, *lb_changed* will be true for at most $\delta(G) + k$ invocations of CTCP, as the maximum k-plex size is upper bounded by $\delta(G)+k$. Thus, the total time complexity of running Lines 3–4 of Algorithm 3 for all invocations to CTCP is $O((\delta(G)+k) \times |E|) = O(\delta(G) \times |E|)$ as k is bounded by a small constant.

Secondly, each edge (u, v) of G will be popped out from Q_e at Line 10 of Algorithm 3 at most once. For each edge (u, v) popped from Q_e, the time complexity of running Lines 13–16 of Algorithm 3 is $O(\min\{d_G(u), d_G(v)\})$ by using hash-based set intersection. When summing over all edges of G, the total time complexity of running Lines 13–16 of Algorithm 3 for all invocations to truss-peeling is $O(\sum_{(u,v)\in E} \min\{d_G(u), d_G(v)\}) = O(\delta(G) \times |E|)$.

Thirdly, each vertex of G will be popped out from Q_v at Line 18 of Algorithm 3 at most once. For each vertex u popped from Q_v, the time complexity of running Lines 19–26 is $O((\deg(u))^2)$ where $\deg(u) \le \delta(G)+k$, as $\deg(u) \le \delta(G)$ at Line 15 of Algorithm 1 and $\deg(u) \le \tau_v \le \delta(G) + k$ at line 6 of Algorithm 3; here, we assume a hash-structure is built for checking the existence of an edge at Line 23 of Algorithm 3. Thus, the total time complexity of running Lines 19–26 of Algorithm 3 for all invocations to truss-peeling is $O(\delta(G) \times |E|)$.

Note that, to efficiently obtain a vertex with degree smaller than τ_v at Line 6 of Algorithm 3, we resort to the bin sort-like data structure that is used in core decomposition algorithms and achieves this in amortized constant time. Thus, the theorem follows.

A.2 Proof of Lemma 5.2

Let's consider any k-plex P in g such that $S \subseteq P$. Then $P \setminus S$ can be partitioned into four disjoint subsets as shown in Figure 6: (1) N^{cn}, common neighbors of u and v; (2) $N^{e}(u)$, exclusive neighbors of u (i.e., $N_{P\setminus S}(u) \setminus N^{cn}$); (3) $N^{e}(v)$, exclusive neighbors of v; and (4) N^{cnon}, common non-neighbors of u and v. By the definitions of $r_S(u)$ and $r_S(v)$, we have

$$|N^{e}(v)| + |N^{cnon}| \leq r_S(u) \quad \text{and} \quad |N^{e}(u)| + |N^{cnon}| \leq r_S(v)$$

as all vertices in $N^{e}(v) \cup N^{cnon}$ are not adjacent to u. Consequently, $|P| = |S| + |N^{e}(u)| + |N^{e}(v)| + |N^{cnon}| + |N^{cn}| \leq |S| + r_S(u) + r_S(v) + \mathsf{cn}_{\overline{S}}(u, v)$, as $|N^{cn}| \leq \mathsf{cn}_{\overline{S}}(u, v)$.