Efficient Enumeration of Large Maximal k-Plexes

Qihao Cheng*
Tsinghua University
cqh22@mails.tsinghua.edu.cn

Da Yan*
Indiana University Bloomington
yanda@iu.edu

Tianhao Wu*
Tsinghua University
wuth20@mails.tsinghua.edu.cn

Lyuheng Yuan
Indiana University Bloomington
lyyuan@iu.edu

Ji Cheng
HKUST
jchengac@connect.ust.hk

Zhongyi Huang
Tsinghua University
zhongyih@tsinghua.edu.cn

Yang Zhou
Auburn University
yangzhou@auburn.edu

ABSTRACT

Finding cohesive subgraphs in a large graph has many important applications, such as community detection and biological network analysis. Clique is often a too strict cohesive structure since communities or biological modules rarely form as cliques for various reasons such as data noise. Therefore, k-plex is introduced as a popular clique relaxation, which is a graph where every vertex is adjacent to all but at most k vertices. In this paper, we propose a fast branch-and-bound algorithm as well as its task-based parallel version to enumerate all maximal k-plexes with at least q vertices. Our algorithm adopts an effective search space partitioning approach that provides a lower time complexity, a new pivot vertex selection method that reduces candidate vertex size, an effective upper-bounding technique to prune useless branches, and three novel pruning techniques by vertex pairs. Our parallel algorithm uses a timeout mechanism to eliminate straggler tasks, and maximizes cache locality while ensuring load balancing. Extensive experiments show that compared with the state-of-the-art algorithms, our sequential and parallel algorithms enumerate large maximal k-plexes with up to 5× and 18.9× speedup, respectively. Ablation results also demonstrate that our pruning techniques bring up to 7× speedup compared with our basic algorithm. Our code is released at https://github.com/chengqihao/Maximal-kPlex.

1 INTRODUCTION

Finding cohesive subgraphs in a large graph is useful in various applications, such as finding protein complexes or biologically relevant functional groups [5, 11, 22, 37], and social communities [21, 28] that can correspond to cybercriminals [41], botnets [35, 41] and spam/phishing email sources [34, 40]. One classic notion of cohesive subgraph is *clique* which requires every pair of distinct vertices to be connected by an edge. However, in real graphs, communities rarely appear in the form of cliques due to various reasons such as the existence of data noise [14, 15, 31, 39].

As a relaxed clique model, k-plex was first introduced in [33], which is a graph where every vertex is adjacent to all but at most k vertices. It has found extensive applications in the analysis of social networks [33], especially in the community detection [15, 31]. However, mining k-plexes is NP-hard [6, 27], so existing algorithms rely on branch-and-bound search which runs in exponential time in the worst case. Many recent works have studied the branch-and-bound algorithms for mining maximal k-plexes [15, 16, 39, 47] and finding a maximum k-plex [12, 18, 25, 42, 46], with various techniques proposed to prune the search space. We will review these works in Section 2.

In this paper, we study the problem of enumerating all maximal k-plexes with at least q vertices (and are hence large and statistically significant), and propose a more efficient branch-and-bound algorithm with new search-space pruning techniques and parallelization techniques to speed up the computation. Our search algorithm treats each set-enumeration subtree as an independent task, so that different tasks can be processed in parallel. Our main contributions are as follows:

- We propose a method for search space partitioning to create independent searching tasks, and show that its time complexity is $O\left(nr_1^k r_2 \gamma_k^D\right)$, where n is the number of vertices, D is the graph degeneracy, and let Δ be the maximum degree, then $r_1 = \min\left\{\frac{D\Delta}{q-2k+2}, n\right\}$, $r_2 = \min\left\{\frac{D\Delta^2}{q-2k+2}, nD\right\}$, and $\gamma_k < 2$ is a constant close to 2.
- We propose a new approach to selecting a pivot vertex to expand the current k-plex by maximizing the number of saturated vertices (i.e., those vertices whose degree is the minimum allowed to form a valid k-plex) in the k-plex. This approach effectively reduces the number of candidate vertices to expand the current k-plex.
- We design an effective upper bound on the maximum size of any k-plex that can be expanded from the current k-plex P, so that if this upper bound is less than the user-specified size threshold, then the entire search branch originated from P can be pruned.
- We propose three novel effective pruning techniques by vertex pairs, and they are integrated into our algorithm to further prune the search space.
- We propose a task-based parallel computing approach over our algorithm to achieve ideal speedup, integrated with a timeout mechanism to eliminate straggler tasks.
- We conduct comprehensive experiments to verify the effectiveness of our techniques, and to demonstrate our superior performance over existing solutions.

The rest of this paper is organized as follows. Section 2 reviews the related work, and Section 3 defines our problem and presents some basic properties of k-plexes. Then, Section 4 describes the branch-and-bound framework of our mining algorithm, Section 5

*The first three authors contributed equally to this work.

further describes the search space pruning techniques, and Section 6 presents our task-based parallelization approach. Finally, Section 7 reports our experiments, and Section 8 concludes this paper.

2 RELATED WORK

Maximal k-Plex Finding. The Bron-Kerbosch (BK) algorithm [10] is a backtracking search algorithm to enumerate maximal cliques, and can be extended to enumerate maximal k-plexes (see Section 4 for details). Many BK-style algorithms are proposed with various effective search space pruning techniques. Specifically, D2K [15] proposes a simple pivoting technique to cut useless branches, which generalizes the pivoting technique for maximal clique finding. A more effective pivoting technique is found by FaPlexen [47]. More recently, FP [16] adopts a new pivoting technique and uses upper-bound-based pruning for maximal k-plex enumeration, but the time complexity is still the same as previous works [15, 47], which is improved by our current work. ListPlex [39] adopts the sub-tasking scheme that partitions the search space efficiently, but it uses the less effective pivoting and branching schemes of FaPlexen, which is avoided in our current work. None of the works has considered effective vertex-pairs pruning techniques proposed in this paper.

Besides BK, maximal k-plexes can also enumerated by a reverse search framework [8]. The key insight is that given a valid k-plex P, it is possible to find another valid one by excluding some existing vertices from and including some new ones to P. Starting from an initial solution, [8] conducts DFS over the solution graph to enumerate all solutions. While the algorithm provides a polynomial delay (i.e., time of waiting for the next valid k-plex) so that it is guaranteed to find some solutions in bounded time, it is less efficient than BK when the goal is to enumerate all maximal k-plexes. Reverse search has also been adapted to work with bipartite graphs [43].

Maximum k-Plex Finding. Conte et al. [14] notice that any node of a k-plex P with $|P| \geq q$ is included in a clique of size at least $\lceil q/k \rceil$, which is used to prune invalid nodes. However, it is necessary to enumerate all maximal cliques which is expensive per se. To find a maximum k-plex, [14] uses binary search to guess the maximum k-plex size for vertex pruning, and then mines k-plexes on the pruned graph to see if such a maximum k-plex can be found, and if not, the maximum k-plex size threshold is properly adjusted for another round of search. However, this approach may fail for multiple iterations before finding a maximum k-plex, so is less efficient than the branch-and-bound algorithms.

BS [42] pioneers a number of pruning techniques in the brand-and-bound framework for finding a maximum k-plex, including the pivoting technique of FaPlexen [47]. In maximum k-plex finding, if the current maximum k-plex is P, then we can prune any branch that cannot generate a k-plex with at least $|P| + 1$ vertices (i.e., upper bound $\leq |P| + 1$). BnB [18] proposes upper bounds and pruning techniques based on deep structural analysis, KpLeX [25] proposes an upper bound based on vertex partitioning, and Maplex [46] proposes an upper bound based on graph coloring which is later improved by RGB [45]. kPlexS [12] proposes a CTCP technique to prune the vertices and edges using the second-order property, and it shows that the reduced graph by CTCP is guaranteed to be no larger than that computed by BnB, Maplex and KpLeX. kPlexS also proposed new techniques for branching and pruning, and outperforms BnB, Maplex and KpLeX. In [24, 38], to find maximum k-plexes, an algorithm for the d-BDD problem is applied and a refined upper bound is proposed.

Table 1: List of Important Notations

Notation	Description
P, C, and X	the current k-plex, candidate set, and exclusive set
S	a subset of $N^2_{G_i}(v_i)$
$T_{v_i \cup S}$	a sub-task for set-enumeration search
η	the degeneracy ordering of G, $\{v_1, v_2, \ldots, v_n\}$
$V_{<\eta}(v_i)$, and $V_{\geq \eta}(v_i)$	$\{v_1, v_2, \ldots, v_{i-1}\}$, and $\{v_i, v_2, \ldots, v_n\}$
G_i	the subgraph induced by vertices in $V_{\geq \eta}(v_i)$ within 2 hops from v_i
P_S, and C_S	a sub-task with $P = \{v_i\} \cup S$, and its candidate set
P_m	a maximum k-plex containing the current k-plex P
$supp(v)$	the maximum # of v's non-neighbors outside P that can be added to P
$ub(P)$	the upper bound of the maximum k-plex that P can extend to

Other Dense Subgraphs. There are other definitions of dense subgraphs. Specifically, [13] finds subgraphs to maximize the average degree [13] solvable by a flow-based algorithm, [17] finds the k-vertex subgraph with the most edges, [4] finds k-vertex subgraphs with at least $f(k)$ edges for an edge-density function $f(.)$, and [36] proposes a density measure based on edge surplus to extract a higher-quality subgraph called optimal quasi-clique. However, those problems are very expensive and solved by approximate algorithms, while we target exact k-plex solutions.

Besides k-plex, γ-quasi-clique is the other popular type of clique relaxation whose exact algorithms gained a lot of attention. Branch-and-bound algorithms Crochet [23, 32], Cocain [44], and Quick [29] mine maximal γ-quasi-cliques exactly, and parallel and distributed algorithms have also been developed by our prior works [19, 20, 26]. Unlike k-plex where the restriction at each vertex is on the absolute number of missing edges allowed, γ-quasi-clique places this restriction on the ratio of missing edges (i.e., $(1 - \alpha)$ fraction) at each vertex. This difference makes γ-quasi-clique not satisfying the hereditary property as in k-plexes and cliques [30], making the BK algorithm not applicable, so more expensive branch-and-bound algorithms with sophisticated pruning rules to check are needed.

3 PROBLEM DEFINITION

For ease of presentation, we first define some basic notations. More notations will be defined in Sections 4 and 5 when describing our algorithm, and Table 1 lists the important notations for quick lookup.

Notations. We consider an undirected and unweighted simple graph $G = (V, E)$, where V is the set of vertices, and E is the set of edges. We let $n = |V|$ and $m = |E|$ be the number of vertices and the number of edges, respectively. The diameter of G, denoted by $\delta(G)$, is the shortest-path distance of the farthest pair of vertices in G, measured by the # of hops.

For each vertex $v \in V$, we use $N^c_G(v)$ to denote the set of vertices with distance exactly c to v in G. For example, $N^1_G(v)$ is v's direct neighbors in G, which we may also write as $N_G(v)$; and $N^2_G(v)$ is the set of all vertices in G that are 2 hops away from v. The degree of a vertex v is denoted by $d_G(v) = |N_G(v)|$, and the maximum vertex degree in G is denoted by Δ.

We also define the concept of *non-neighbor*: a vertex u is a non-neighbor of v in G if $(u, v) \notin E$. Accordingly, the set of non-neighbors of v is denoted by $\overline{N_G}(v) = V - N_G(v)$, and we denote its cardinality by $\overline{d_G}(v) = |\overline{N_G}(v)|$.

Given a vertex subset $S \subseteq V$, we denote by $G[S] = (S, E[S])$ the subgraph of G induced by S, where $E[S] = \{(u, v) \in E \mid u \in S \wedge v \in S\}$. We simplify the notation $N_{G[S]}(v)$ to $N_S(v)$, and define other notations such as $\overline{N_S}(v)$, $d_S(v)$, $\overline{d_S}(v)$ and $\delta(S)$ in a similar manner.

The k-core of an undirected graph G is its largest induced subgraph with minimum degree k. The degeneracy of G, denoted by D, is the largest value of k for which a k-core exists in G. The degeneracy of a graph may be computed in linear time by a peeling algorithm that repeatedly removes the vertex with the minimum current degree at a time [7], which produces a degeneracy ordering of vertices denoted by $\eta = [v_1, v_2, \ldots, v_n]$. All the consecutively removed vertices with the minimum current degree being k ($k = 0, 1, \cdots, D$) constitute a k-shell, and in degeneracy ordering, vertices are listed in segments of k-shells with increasing k. We order vertices in the same k-shell by vertex ID (from the input dataset) to make η unique, though our tests by shuffling within-shell vertex ordering show that it has a negligible impact on the time difference for our k-plex mining. In a real graph, we usually have $D \ll n$.

Problem Definition. We next define our mining problem. As a relaxed clique model, a k-plex is a subgraph $G[P]$ that allows every vertex u to miss at most k links to vertices of P (including u itself), i.e., $d_P(u) \geq |P| - k$ (or, $\overline{d_P}(u) \leq k$):

Definition 3.1. (k-*Plex*) Given an undirected graph $G = (V, E)$ and a positive integer k, a set of vertices $P \subseteq V$ is a k-plex iff for every $u \in P$, its degree in $G[P]$ is no less than $(|P| - k)$.

Note that k-plex satisfies the hereditary property:

Theorem 3.2. (Hereditariness) *Given a k-plex $P \subseteq V$, any subset $P' \subseteq P$ is also a k-plex.*

This is because for any $u \in P'$, we have $u \in P$ and since P is a k-plex, $\overline{d_P}(u) = |\overline{N_P}(u)| \leq k$. Since $\overline{N_{P'}}(u) \subseteq \overline{N_P}(u)$, we have $\overline{d_{P'}}(u) = |\overline{N_{P'}}(u)| \leq k$, so P' is also a k-plex.

Another important property is that if a k-plex P satisfies $|P| > 2k - c$, then $G[P]$ is connected with the diameter $\delta(P) \leq c$ ($c \geq 2$) [42]. A common assumption by existing works [15, 16] is the special case when $c = 2$:

Theorem 3.3. *Given a k-plex P, if $|P| \geq 2k - 1$, then $\delta(P) \leq 2$.*

This is a reasonable assumption since natural communities that k-plexes aim to discover are connected, and we are usually interested in only large (hence statistically significant) k-plexes with size at least q. For $k \leq 5$, we only require $q \geq 2k - 1 = 9$. Note that a k-plex with $|P| = 2k - 2$ may be disconnected, such as one formed by two disjoint $(k - 1)$-cliques.

A k-plex is said to be maximal if it is not a subgraph of any larger k-plex. We next formally define our problem:

Definition 3.4. (*Size-Constrained Maximal k-Plex Enumeration*) Given a graph $G = (V, E)$ and an integer size threshold $q \geq 2k-1$, find all the maximal k-plexes with at least q vertices.

Note that instead of mining G directly, we can shrink G into its $(q-k)$-core for mining, which can be constructed in $O(m+n)$ time using the peeling algorithm that keeps removing those vertices with degree less than $(q - k)$:

Theorem 3.5. *Given a graph $G = (V, E)$, all the k-plexes with at least q vertices must be contained in the $(q - k)$-core of G.*

This is because for any vertex v in a k-plex P, $d_P(v) \geq |P| - k$, and since we require $|P| \geq q$, we have $d_P(v) \geq q - k$.

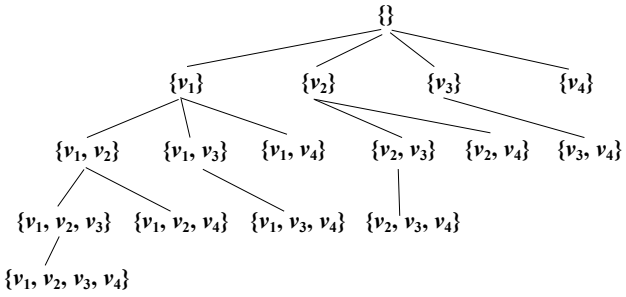


Figure 1: Set-Enumeration Search Tree

Algorithm 1: *bron_kerbosch*(P, C, X)

```
1 if C = ∅ and X = ∅ do output P, return
2 foreach vertex v_i ∈ C do
3     P' ← P ∪ {v_i}, C ← C − {v_i}
4     C' ← {u | u ∈ C ∧ P' ∪ {u} is a k-plex}
5     X' ← {u | u ∈ X ∧ P' ∪ {u} is a k-plex}
6     bron_kerbosch(P', C', X')
7     X ← X ∪ {v_i}
```

4 BRANCH-AND-BOUND ALGORITHM

This section describes the branch-and-bound framework of our mining algorithm. Section 5 will further describe the pruning techniques that we use to speed up our algorithm.

Set-Enumeration Search. Figure 1 shows the set-enumeration tree T for a graph G with four vertices $V = \{v_1, v_2, v_3, v_4\}$ where we assume vertex order $v_1 < v_2 < v_3 < v_4$. Each tree node represents a vertex set P, and only those vertices larger than the largest vertex in P are used to extend P. For example, in Figure 1, node $\{v_1, v_3\}$ can be extended with v_4 but not v_2 since $v_2 < v_3$; in fact, $\{v_1, v_2, v_3\}$ is obtained by extending $\{v_1, v_2\}$ with v_3. Let us denote T_P as the subtree of T rooted at a node with set P. Then, T_P represents a search space for all possible k-plexes that contain all the vertices in P. We represent the task of mining T_P as a pair $\langle P, C\rangle$, where P is the set of vertices assumed to be already included, and $C \subseteq (V - P)$ keeps those vertices that can extend P further into a valid k-plex. The task of mining T_P, i.e., $\langle P, C\rangle$, can be recursively decomposed into tasks that mine the subtrees rooted at the children of P in T_P.

Algorithm 1 describes how this set-enumeration search process is generated, where we first ignore the red parts, and begin by calling *bron_kerbosch*$(P = \emptyset, C = V)$. Specifically, in each iteration of the for-loop in Lines 2–7, we consider the case where v_i is included into P (see P' in Lines 3 and 6). Here, Line 4 is due to the hereditary property: if $P' \cup \{u\}$ is not a k-plex, then any superset of $P' \cup \{u\}$ cannot be a k-plex, so $u \notin C'$. Also, Line 3 removes v_i from C so in later iterations, v_i is excluded from any subgraph grown from P.

Note that while the set-enumeration tree in Figure 1 ensures no redundancy, i.e., every subset of V will be visited at most once, it does not guarantee set maximality: even if $\{v_1, v_2, v_4\}$ is a k-plex, $\{v_2, v_4\}$ will still be visited but it is not maximal.

Bron-Kerbosch Algorithm. The Bron-Kerbosch algorithm as shown in Algorithm 1 avoids outputting non-maximal k-plexes with the help of an exclusive set X. The algorithm was originally

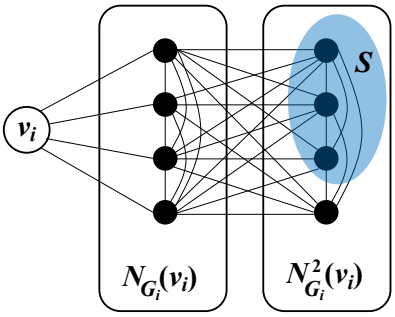


Figure 2: Decomposition of Top-Level Task T_{v_i}

proposed to mine maximal cliques [10], and has been recently adapted for mining maximal k-plexes [15, 47].

Specifically, after each iteration of the for-loop where since we consider the case with v_i included into P, we add v_i to X in Line 7 so that in later iterations (where v_i is not considered for extending P), v_i will be used to check result maximality.

We can redefine the task of mining T_P as a triple $\langle P, C, X\rangle$ with three disjoint sets, where the exclusive set X keeps all those vertices that have been considered before (i.e., added by Line 7), and can extend P to obtain larger k-plexes (see Line 5 which refines X into X' based on P'). Those k-plexes should have been found before.

When there is no more candidate to grow P (i.e., $C = \emptyset$ in Line 1), if $X \neq \emptyset$, then based on Line 5, $P \cup \{u\}$ is a k-plex for any $u \in X$, so P is not maximal. Otherwise, P is maximal (since such a u does not exist) and outputted. For example, let $P = \{v_2, v_4\}$ and $X = \{v_1\}$, then we cannot output P since $\{v_1, v_2, v_4\} \supseteq P$ is also a k-plex, so P is not the maximal one.

Initial Tasks. Referring to Figure 1 again, the top-level tasks are given by $P = \{v_1\}$, $\{v_2\}$, $\{v_3\}$ and $\{v_4\}$, which are generated by *bron_kerbosch*$(P = \emptyset, C = V, X = \emptyset)$. It is common to choose the precomputed degeneracy ordering $\eta = [v_1, v_2, \ldots, v_n]$ to conduct the for-loop in Line 2, which was found to generate more load-balanced tasks $T_{\{v_i\}}$ [15, 16, 39, 47]. Intuitively, each vertex v_i is connected to at most D vertices among later candidates $\{v_{i+1}, v_{i+2}, \ldots, v_n\}$ based on the peeling process, and D is usually a small value.

Note that we do not need to mine each $T_{\{v_i\}}$ over the entire G. Let us define $V_{<\eta}(v_i) = \{v_1, v_2, \ldots, v_{i-1}\}$ and $V_{\geq\eta}(v_i) = \{v_i, v_{i+1}, v_{i+2}, \ldots, v_n\}$, then we only need to mine $T_{\{v_i\}}$ over

$$G_i = G\left[V_{\geq\eta}(v_i) \cap \left(\{v_i\} \cup N_G(v_i) \cup N^2_G(v_i)\right)\right], \qquad (1)$$

since candidates in C must be after v_i in η, and must be within two hops from v_i according to Theorem 3.3. In fact, since G_i tends to be dense, it is efficient when G_i is represented by an adjacency matrix [12]. We call v_i as a seed vertex, and call G_i as a seed subgraph.

As a further optimization, we decompose $T_{\{v_i\}}$ into disjoint sub-tasks $T_{\{v_i\}\cup S}$ for subsets $S \subseteq N^2_{G_i}(v_i)$, where the vertices of S are the only vertices in $N^2_{G_i}(v_i)$ allowed to appear in a k-plex found in $T_{\{v_i\}\cup S}$, and other candidates have to come from $N_{G_i}(v_i)$ (see Figure 2). We only need to consider $|S| < k$ since otherwise, v_i has at least k non-neighbors $S \subseteq N^2_{G_i}(v_i)$, plus v_i itself, v_i misses $(k+1)$ edges which violates the k-plex definition, so $\{v_i\} \cup S$ cannot be a k-plex, neither can its superset due to the hereditary property.

Algorithm 2: Enumerating k-Plex with Initial Tasks

Input: Graph $G = (V, E)$, k, $q \geq 2k - 1$

1 $G \leftarrow$ the $(q-k)$-core of G // Using Theorem 3.5
2 $\eta = \{v_1, \ldots, v_n\}$ is the degeneracy ordering of V
3 **for** $i = 1, 2, \ldots, n - q + 1$ **do**
4 $\quad V_i \leftarrow \{v_i, v_{i+1}, \ldots, v_n\} \cap \left(\{v_i\} \cup N_G(v_i) \cup N^2_G(v_i)\right)$
5 $\quad V'_i \leftarrow \{v_1, v_2, \ldots, v_{i-1}\} \cap \left(N_G(v_i) \cup N^2_G(v_i)\right)$
6 $\quad G_i \leftarrow G[V_i]$, and apply further pruning over G_i
7 $\quad$ **foreach** $S \subseteq N^2_{G_i}(v_i)$ *that* $|S| \leq k - 1$ **do**
8 $\quad\quad P_S \leftarrow \{v_i\} \cup S, \quad C_S \leftarrow N_{G_i}(v_i)$
9 $\quad\quad X_S \leftarrow V'_i \cup (N^2_{G_i}(v_i) - S)$
10 $\quad\quad$ Call Branch$(G_i, k, q, P_S, C_S, X_S)$

In summary, each search tree $T_{\{v_i\}}$ creates a *task group* sharing the same graph G_i (c.f. Eq (1)), where each task mines the search tree $T_{\{v_i\}\cup S}$ for a subset $S \subseteq N^2_{G_i}(v_i)$ with $|S| < k$.

Algorithm 2 shows the pseudocode for creating initial task groups, where in Line 1 we shrink G into its $(q-k)$-core by Theorem 3.5, so n is reduced. Line 2 then orders the vertices of G in degeneracy order to keep the size of all G_i small to generate more load-balanced tasks by bounding the candidate size $|C|$. This ordering is also essential for our time complexity analysis in Section 5.

We then generate initial task groups $T_{\{v_i\}}$ using the for-loop from Line 3, where we skip $i > n - q + 1$ since $|V_i| < q$ in this case; here, V_i is the vertex set of G_i (see Lines 4 and 6). For each task $T_{\{v_i\}\cup S} = \langle P, C, X\rangle$ of the task group $T_{\{v_i\}}$, we have $P = \{v_i\} \cup S$ and $C \subseteq N_{G_i}(v_i) \triangleq C_S$ (Line 8). Let us define $N^{1,2}_{<\eta}(v_i) = V_{<\eta}(v_i) \cap \left(N_G(v_i) \cup N^2_G(v_i)\right)$ (i.e., V'_i in Line 5), then we also have $X \subseteq N^{1,2}_{<\eta}(v_i) \cup (N^2_{G_i}(v_i) - S) \triangleq X_S$ (Line 9). This is because vertices of $(N^2_{G_i}(v_i) - S)$ may be considered by other sub-tasks $T_{\{v_i\}\cup S'}$, and vertices of $N^{1,2}_{<\eta}(v_i)$ may be considered by other task groups $T_{\{v_j\}}$ $(j < i)$ (if v_j is more than 2 hops away from v_i, it cannot form a k-plex with v_i, so v_j is excluded from X_S). Line 7 of Algorithm 2 is implemented by the set-enumeration search of S over $N^2_G(v_i)$, similar to Algorithm 1 Lines 2, 3 and 6.

Finally, to maintain the invariant of Bron-Kerbosch algorithm (c.f., Lines 4–5 of Algorithm 1), we set $C \leftarrow \{u \mid u \in C_S \wedge P \cup \{u\}$ is a k-plex$\}$ and $X \leftarrow \{u \mid u \in X_S \wedge P \cup \{u\}$ is a k-plex$\}$, and mine $T_{\{v_i\}\cup S} = \langle P, C, X\rangle$ recursively using the Bron-Kerbosch algorithm of Algorithm 1 over G_i. Instead of directly running Algorithm 1, we actually run a variant to be described in Algorithm 3 which applies more pruning techniques, and refines C_S and X_S into C and X, respectively, at the very beginning. This branch-and-bound sub-procedure is called in Line 10 of Algorithm 2.

Branch-and-Bound Search. Algorithm 3 first updates C and X to ensure that each vertex in C or X can form a k-plex with P (Lines 2–3). If $C = \emptyset$ (Line 2), there is no more candidate to expand P with, so Line 6 returns. Moreover, if $X = \emptyset$ (i.e., P is maximal) and $|P| \geq q$, we output P (Line 5).

Otherwise, we pick a pivot v_p (Lines 7–10 and 15–16) and compute an upper bound ub of the maximum size of any k-plex that $P \cup \{v_p\}$ may expand to (Line 17). The branch expanding $P \cup \{v_p\}$ is filtered if $ub < q$ (Lines 18–19), while the branch

Algorithm 3: Branch-and-Bound Search

```
1  function Branch(G, k, q, P, C, X)
2      C ← {v | v ∈ C ∧ P ∪ {v} is a k-plex}
3      X ← {v | v ∈ X ∧ P ∪ {v} is a k-plex}
4      if C = ∅ then
5          if X = ∅ and |P| ≥ q then    Output P
6          return
7      M_0 ← the subset of P ∪ C with minimum degree in
         G[P ∪ C]
8      M ← the subset of M_0 with maximum d̄_P(v)
9      if M ∩ P ≠ ∅ then    Pick a pivot v_p ∈ M ∩ P
10     else    Pick a pivot v_p ∈ M ∩ C
11     if d_{P∪C}(v_p) ≥ |P| + |C| − k then
12         if P ∪ C is a maximal k-plex then
13             if |P ∪ C| ≥ q then    Output P ∪ C
14         return
15     if v_p ∈ P then
16         Re-pick a pivot v_new from N̄_C(v_p) using the same
             rules as in Lines 7–10;    v_p ← v_new
17     Compute the upper bound ub of the size of any k-plex
         that can be expanded from P ∪ {v_p}
18     if ub ≥ q then
19         Branch(G, k, q, P ∪ {v_p}, C − {v_p}, X)
20     Branch(G, k, q, P, C − {v_p}, X ∪ {v_p})
```

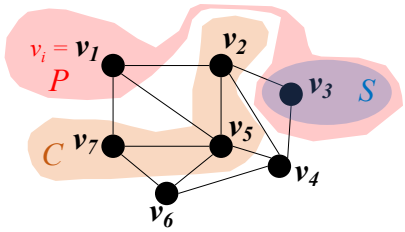


Figure 3: A Toy Graph for Illustration

excluding v_p is always executed in Lines 20. We explain how ub is computed later.

Pivot Selection. We next explain our pivot selection strategy. Specifically, Lines 7–10 select $v_p \in P \cup C$ to be a vertex with the minimum degree in $G[P \cup C]$, so that in Line 11, if $d_{P\cup C}(v_p) \geq |P| + |C| - k$, then for any other $v \in P \cup C$, we have $d_{P\cup C}(v) \geq d_{P\cup C}(v_p) \geq |P| + |C| - k$, and hence $P \cup C$ is a k-plex that we then examine for maximality. In this case, we do not need to expand further so Line 14 returns. In Line 12, we check if $P \cup C$ is maximal by checking if $\{v \mid v \in X \wedge P \cup C \cup \{v\}$ is a k-plex$\}$ is empty.

Note that among those vertices with the minimum degree in $G[P \cup C]$, we choose v_p with the maximum $\overline{d_P}(v)$ (Line 8) which tends to prune more candidates in C. Specifically, if $\overline{d_P}(v_p) = k$ and v_p is in (or added to) P, then v_p's non-neighbors in C are pruned; such a vertex v_p is called *saturated*.

Note that *if more saturated vertices are included in P, more vertices in C tend to be pruned.* We, therefore, pick a pivot to maximize the number of saturated vertices in P. Specifically, we try to find the closest-to-saturation pivot v_p in P (Line 9), and then find a non-neighbor of v_p in C that is closer to saturation (Lines 16) as the new pivot $v_{new} \in C$, which is then used to expand P (Line 19). While if the closest-to-saturation pivot v_p cannot be found in P, we then pick v_p in C (Line 10), which is then used to expand P (Line 19). Example 4.1 illustrates the process of our pivot selection strategy.

Example 4.1 (Pivot Selection). Consider the graph G shown in Figure 3 and $k = 2$. Assume that the current k-plex $P = \{v_1, v_3\}$ and the candidate set $C = \{v_2, v_5, v_7\}$. Then according to Lines 7–8, $M_0 = \{v_3\}$ and $M = \{v_3\}$. Note that $M \cap P = \{v_3\} \neq \emptyset$, so $v_p = v_3$. Following our re-picking strategy, the new pivot vertex is selected from $\overline{N_C}(v_3) = \{v_5, v_7\}$, and the selected pivot vertex is v_7.

5 UPPER BOUNDING AND PRUNING

This section introduces our upper bounding and additional pruning techniques used in Algorithms 2 and 3, respectively, that are critical in speeding up the enumeration process. Due to space limitation, we put most of the proofs in the appendix of our full version [3].

Seed Subgraph Pruning. The theorem below states the second-order property of two vertices in a k-plex with size constraint:

Theorem 5.1. *Let P be a k-plex with $|P| \geq q$. Then, for any two vertices $u, v \in P$, we have (i) if $(u, v) \notin E$, $|N_P(u) \cap N_P(v)| \geq q - 2k + 2$, (ii) otherwise, $|N_P(u) \cap N_P(v)| \geq q - 2k$.*

Proof. Please see Appendix A.1 [3]. □

Note that by setting $q = 2k-1$, Case (i) gives $|N_P(u) \cap N_P(v)| \geq (2k-1) - 2k + 2 = 1$, i.e., for any two vertices $u, v \in P$ that are not mutual neighbors, they must share a neighbor and is thus within 2 hops, which proves Theorem 3.3.

This also gives the following corollary to help further prune the size of a seed subgraph G_i in Line 6 of Algorithm 2, which is also essential for our time complexity analysis (c.f., Lemma 5.9).

Corollary 5.2. *Consider an undirected graph $G = (V, E)$ and an ordering of V: $\{v_1, v_2, .., v_n\}$. Let v_i be the seed vertex and G_i be the seed subgraph. Then, any vertex $u \in V_i$ (recall Algorithm 2 Line 4 for the definition of V_i) that satisfies either of the following two conditions can be pruned:*

- *$u \in N_{G_i}(v_i)$ and $|N_{G_i}(u) \cap N_{G_i}(v_i)| < q - 2k$;*
- *$u \in N^2_{G_i}(v_i)$ and $|N_{G_i}(u) \cap N_{G_i}(v_i)| < q - 2k + 2$.*

Upper Bound Computation. We next consider how to obtain the upper bound of the maximum size of a k-plex containing P, which is called in Algorithm 3 Line 17.

Theorem 5.3. *Given a k-plex P in a seed subgraph $G_i = (V_i, E_i)$, the upper bound of the maximum size of a k-plex containing P is $\min_{u\in P}\{d_{G_i}(u)\} + k$.*

Proof. We illustrate the proof process using Figure 4 (where irrelevant edges are omitted). Let $P_m \subseteq P \cup C$ be a maximum k-plex containing P. Given any $u \in P$, we can partition P_m into two sets: (1) $N_{G_i}(u) \cap P_m$ (see the red vertices in Figure 4), and (2) $\overline{N_{G_i}}(u) \cap P_m$ (see the green vertices in Figure 4 including u itself). The first set $N_{G_i}(u) \cap P_m$ has size at most $|N_{G_i}(u)| = d_{G_i}(u)$ (i.e., at most all the 7 neighbors in $N_{G_i}(u)$ in Figure 4 are included into P_m). For the second set, if more than k vertices are included into P_m, then $u \in P_m$ is a non-neighbor of k vertices in P_m, so P_m violates the k-plex definition (Definition 3.1) which leads to a contradiction; as a result, at most k vertices in $\overline{N_{G_i}}(u) \cap P_m$ (including

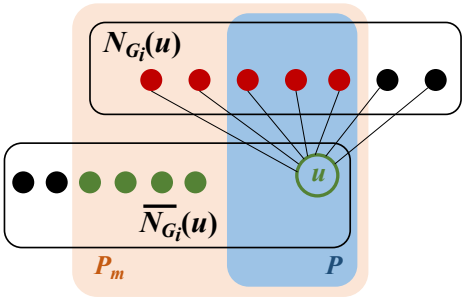


Figure 4: Upper Bound Illustration for Theorem 5.3

u itself already in P_m) can be added to P_m. Putting things together, $|P_m| = |N_{G_i}(u) \cap P_m| + |\overline{N_{G_i}}(u) \cap P_m| \leq d_{G_i}(u) + k$. Since u can be an arbitrary vertex in P, we have $|P_m| \leq \min_{u \in P}\{d_{G_i}(u)\} + k$. □

We use our running example to illustrate how to use Theorem 5.3.

Example 5.4. Consider the graph G shown in Figure 3 and $k = 2$, and assume that $P = \{v_1, v_3\}$. For v_1, we can add at most all its 3 neighbors $\{v_2, v_5, v_7\}$ and $k = 2$ non-neighbors into a k-plex containing P. Thus, the upper bound of its size is $3 + 2 = 5$. Similarly, for v_3, the upper bound is $2 + 2 = 4$. Therefore, the size of the k-plex expanded from P is at most $\min\{5, 4\} = 4$.

In Line 17 of Algorithm 3, we compute $\min_{u \in P \cup \{v_p\}}\{d_{G_i}(u)\} + k$ as an upper bound by Theorem 5.3. Recall that we already select v_p as the vertex in G_i with the minimum degree in Line 7 of Algorithm 3, so the upper bound can be simplified as $d_{G_i}(v_p) + k$. Note that v_p here is the one obtained in Lines 7–10, not the v_{new} that replaces the old v_p in Line 16 in the case when $v_p \in P$.

We next derive another upper bound of the maximum size of a k-plex containing P. First, we define the concept of "support number of non-neighbors". Given a k-plex P and candidate set C, for a vertex $v \in P \cup C$, its support number of non-neighbors is defined as $\sup_P(v) = k - \overline{d_P}(v)$, which is the maximum number of non-neighbors of v **outside** P that can be included in any k-plex containing P.

Theorem 5.5. *Let v_i and $G_i = (V_i, E_i)$ be the seed vertex and corresponding seed subgraph, respectively, and consider a sub-task $P_S = S \cup \{v_i\}$ where $S \subseteq N^2_{G_i}(v_i)$.*

For a k-plex P satisfying $P_S \subseteq P \subseteq V_i$ and for a pivot vertex $v_p \in C \subseteq C_S = N_{G_i}(v_i)$, the upper bound of the maximum size of a k-plex containing $P \cup \{v_p\}$ is

$$|P| + sup_P(v_p) + |K|, \tag{2}$$

where the set K is computed as follows:

Initially, $K = N_C(v_p)$. For each $w \in K$, we find $u_m \in \overline{N_P}(w)$ such that $sup_P(u_m)$ is the minimum; if $sup_P(u_m) > 0$, we decrease it by 1. Otherwise, we remove w from K.

Figure 5 shows the rationale of the upper bound in Eq (2), where $v_p \in C_S$ is to be added to P (shown inside the dashed contour). Let $P_m \subseteq P \cup C$ be a maximum k-plex containing $P \cup \{v_p\}$, then the three terms in Eq (2) correspond to the upper bounds of the three sets that P_m can take its vertices from: (1) P whose size is exactly $|P|$, (2) v_p's non-neighbors in C, i.e., $P_m \cap \overline{N_C}(v_p)$ (including v_p itself) whose size is upper-bounded by $\sup_P(v_p) = k - \overline{d_P}(v_p)$, and (3) v_p's neighbors in C, i.e., $P_m \cap N_C(v_p)$ whose size is at most $|K|$.

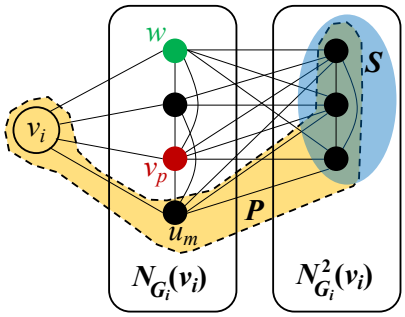


Figure 5: Upper Bound Illustration for Theorem 5.5

Here, K computes the largest set of candidates in $N_C(v_p)$ that can expand $P \cup \{v_p\}$. Specifically, for each $w \in N_C(v_p)$, if there exists a non-neighbor in P, denoted by u_m, that has $\sup_P(u_m) = 0$, then w is pruned (from K) since if we move w to P, u_m would violate the k-plex definition. Otherwise, we decrement $\sup_P(u_m)$ to reflect that w (which is a non-neighbor of u_m) has been added to K (i.e., removed from C) to expand $P \cup \{v_p\}$. Algorithm 4 shows the above approach to compute the upper bound, which is called in Line 17 of Algorithm 3.

Algorithm 4: Computing Upper Bound by Theorem 5.5

```
1 sup_P(v_p) ← k − d̄_P(v_p)
2 foreach u ∈ P do  sup_P(u) ← k − d̄_P(u)
3 ub ← |P| + sup_P(v_p)
4 foreach w ∈ N_C(v_p) do
5     Find u_m ∈ N̄_P(w) s.t. sup_P(u_m) is the minimum
6     if sup_P(u_m) > 0 then
7         sup_P(u_m) ← sup_P(u_m) − 1
8         ub ← ub + 1
9 return ub
```

We next prove that K is truly the largest set of candidates in $N_C(v_p)$ that can expand $P \cup \{v_p\}$.

Proof. Please see Appendix A.2 [3]. □

Putting Theorems 5.3 and 5.5 together, the upper bound in Line 17 of Algorithm 3 is given by

$$\min\{|P| + \sup_P(v_p) + |K|, d_{G_i}(v_p) + k\}. \tag{3}$$

We use our running example to illustrate how to use Theorem 5.5.

Example 5.6. We use the same graph and settings as above two examples, i.e., $k = 2$, $P = \{v_1, v_3\}$, and $C = \{v_2, v_5, v_7\}$. According to Example 4.1, the pivot vertex is v_7. We can calculate $\sup_P(v_7) = k - \overline{d_P}(v_7) = 1$ since v_7 only has one non-neighbor v_3 in P (i.e., $\overline{d_P}(v_7) = 1$). Also, K is initialized as $N_C(v_7) = \{v_5\}$. For $w = v_5$, since $\overline{N_P}(v_5) = \{v_3\}$, we have $u_m = v_3$. Since $\sup_P(v_3) = k - \overline{d_P}(v_3) = 0$ (as v_3 has two non-neighbors $\{v_1, v_3\}$ in P), $w = v_5$ is removed from K, so $K = \emptyset$. Thus, the upper bound of the size of the k-plex expanded from P is $|P| + \sup_P(v_7) + |K| = 2 + 1 + 0 = 3$.

For an initial sub-task $\langle P_S, C_S, X_S \rangle$, we can further improve Theorem 5.5 as follows.

Theorem 5.7. *Let v_i and $G_i = (V_i, E_i)$ be the seed vertex and corresponding seed subgraph, respectively, and consider a sub-task*

$P_S = S \cup \{v_i\}$ *where* $S \subseteq N^2_{G_i}(v_i)$ *and* $S \neq \emptyset$. *We caculate* $|K|$ *with a modified version of Algorithm 4 with* $v_p = v_i$, *and* $\sup_P(v_i) = 0$ *in Line 1. The upper bound is* $|P_S| + \sup_P(v_i) + |K| = |P_S| + |K|$.

Proof. Please see Appendix A.3 [3]. □

Intuitively, this is a special case of Theorem 5.5 with $P = S$ and $v_p = v_i$, and $\sup_P(v_i) = 0$ since there does not exist any non-neighbor of v_i in C to be added to P (recall from Line 8 of Algorithm 2 that C only contains v_i's neighbors).

Recall that Theorem 5.3 also gives another upper bound of the maximum size of a k-plex containing P_S, which is $\min_{v \in P_S}\{d_{G_i}(v)\} + k$. Combining with Theorem 5.7, the new upper bound is given by $ub(P_S) = \min\left\{|P_S| + |K|, \min_{v \in P_S}\{d_{G_i}(v)\} + k\right\}$.

Right before Line 10 of Algorithm 2, we check if $ub(P_S) < q$; if so, we prune this sub-task without calling Branch(.).

Time Complexity Analysis. We now analyze the time complexity of our algorithm, i.e., Algorithm 2. Recall that D is the degeneracy of G, that Δ is the maximum degree of G, and that seed vertices are in the degeneracy ordering of $V, \eta = \{v_1, \ldots, v_n\}$. Therefore, given a seed subgraph $G_i = (V_i, E_i)$ where $V_i \subseteq \{v_i, v_{i+1}, \ldots, v_n\}$, for each $v \in G_i$, we have $d_{G_i}(v) = |N_{G_i}(v)| \leq D$.

Let us first consider the time complexity of Algorithm 4.

Lemma 5.8. *The time complexity of Algorithm 4 is given by*

$$O(k + (k+1)D) \approx O(D).$$

Proof. Please see Appendix A.4 [3]. □

We next bound the number of sub-tasks in each G_i created by Lines 7 and 10 of Algorithm 2.

Lemma 5.9. *Let* v_i *and* $G_i = (V_i, E_i)$ *be the seed vertex and corresponding seed subgraph, respectively. Also, let us abuse the notation* $|N^2_{G_i}(v_i)|$ *to mean the one pruned by Corollary 5.2 in Line 6 of Algorithm 2. Then, we have* $|N_{G_i}(v_i)| \leq D$ *and* $|N^2_{G_i}(v_i)| = O(r_1)$ *where* $r_1 = \min\left\{\frac{D\Delta}{q-2k+2}, n\right\}$.

Also, the number of subsets $S \subseteq N^2_{G_i}(v_i)$ ($|S| \leq k-1$) *is bounded by* $O\left(|N^2_{G_i}(v_i)|^k\right) = O\left(r_1^k\right)$.

Proof. Please see Appendix A.5 [3]. □

We can also bound the time complexity of Algorithm 3:

Lemma 5.10. *Let* v_i *and* $G_i = (V_i, E_i)$ *be the seed vertex and corresponding seed subgraph, respectively. Then, Branch*$(G_i, k, q, P_S, C_S, X_S)$ *(see Line 10 in Algorithm 2) recursively calls the body of Algorithm 4 for* $O(\gamma_k^D)$ *times, where* $\gamma_k < 2$ *is the maximum positive real root of* $x^{k+2} - 2x^{k+1} + 1 = 0$ *(e.g.,* $\gamma_1 = 1.618$, $\gamma_2 = 1.839$, *and* $\gamma_3 = 1.928$*).*

To see this bound, note that Theorem 1 of [16] has proved that the branch-and-bound procedure is called for $O(\gamma_k^{|C|})$ times. In [16], the candidate set C is taken from vertices within two hops away from each seed vertex v_i, so the branch-and-bound procedure is called for $O(\gamma_k^{|C|}) \leq O(\gamma_k^n)$ times. In our case, $C = C_S = N_{G_i}(v_i)$ which is much tighter since $|C| \leq D$, hence the branch-and-bound procedure is called for $O(\gamma_k^{|C|}) \leq O(\gamma_k^D)$ times.

Finally, consider the cost of the recursion body of Algorithm 3. Note that besides $d_P(.)$, we also maintain $d_{G_i}(.)$ for all vertices in G_i, so that Line 7 of Algorithm 3 (the same applies to Line 16) can obtain the vertices with minimum $d_{G_i}(.)$ in $O(|P| + |C|) \approx O(D)$ time. This is because $P \cup C \subseteq P_S \cup C_S$, so $O(|P| + |C|) = O(|P_S| + |C_S|) = O(k + D) \approx O(D)$, as $|P_S| \leq k$ and $|C_S| \leq D$.

As for the tightening of C and X in Lines 2–3 of Algorithm 3 (the same applies to Line 12), the time complexity is $O(|P|(|C| + |X|))$. Specifically, we first compute the set of saturated vertices in P, denoted by P^*. Since we maintain $d_P(.)$, we can find P^* in $O(|P|)$ time by examining if each vertex u has $d_P(u) = |P| - k$. Then, for each vertex $v \in C \cup X$, we do not prune it iff (1) v is adjacent to all vertices in P^*, and meanwhile, (2) $d_P(v) \geq |P \cup \{v\}| - k = |P| + 1 - k$. This takes $O(|P^*|(|C| + |X|)) = O(|P|(|C| + |X|))$ time.

The recursive body takes time $O(|P|(|C| + |X|))$ which is dominated by the above operation. Note that by Lemma 5.8, Algorithm 3 Line 17 takes only $O(D) \approx O(|P| + |C|)$ time, and the time to select pivot (cost dominated by Line 7) also takes only $O(|P| + |C|)$ time.

Now we are ready to present the time complexity of Algorithm 2.

Theorem 5.11. *Given an undirected graph* $G = (V, E)$ *with degeneracy* D *and maximum degree* Δ, *Algorithm 2 lists all the* k*-plexes with size at least* q *within time* $O\left(nr_1^k r_2 \gamma_k^D\right)$, *where* $r_1 = \min\left\{\frac{D\Delta}{q-2k+2}, n\right\}$ *and* $r_2 = \min\left\{\frac{D\Delta^2}{q-2k+2}, nD\right\}$.

Proof. Please see Appendix A.6 [3]. □

Additional Pruning by Vertex Pairs. We next present how to utilize the further property between vertex pairs in G_i to enable three further pruning opportunities, all based on Lemma 5.12 below.

Lemma 5.12. *Given a* k*-plex* P *and candidate set* C, *the upper bound of the maximum size of a* k*-plex containing* P *is*

$$\min_{u,v \in P}\left\{|P| + sup_P(u) + sup_P(v) + |N_u(C) \cap N_v(C)|\right\}$$

Proof. Please see Appendix A.7 [3]. □

Recall that Algorithm 2 Line 7 enumerates set S from the vertices of $N^2_{G_i}(v_i)$. The first pruning rule below checks if two vertices $u_1, u_2 \in N^2_{G_i}(v_i)$ have sufficient common neighbors in C_S, and if not, then u_1 and u_2 cannot occur together in S.

Theorem 5.13. *Let* v_i *be a seed vertex and* $G_i = (V_i, E_i)$ *be the corresponding seed subgraph. For any two vertices* $u_1, u_2 \in N^2_{G_i}(v_i)$, *if either of the following conditions are met*

- $(u_1, u_2) \in E_i$ *and* $|N_{C_S}(u_1) \cap N_{C_S}(u_2)| < q - k - 2 \cdot \max\{k-2, 0\}$,
- $(u_1, u_2) \notin E_i$ *and* $|N_{C_S}(u_1) \cap N_{C_S}(u_2)| < q - k - 2 \cdot \max\{k-3, 0\}$,

then u_1 *and* u_2 *cannot co-occur in a* k*-plex* P *with* $|P| \geq q$.

Proof. Please see Appendix A.8 [3]. □

Similar analysis can be adapted for the other two cases: (1) $u_1 \in N_{v_i}(G_i)$ and $u_2 \in N^2_{v_i}(G_i)$, and (2) $u_1, u_2 \in N_{v_i}(G_i)$, which we present in the next two theorems.

Theorem 5.14. *Let* v_i *be a seed vertex and* $G_i = (V_i, E_i)$ *be the corresponding seed subgraph. For any two vertices* $u_1 \in N^2_{G_i}(v_i)$ *and* $u_2 \in N_{G_i}(v_i)$, *let us define* $C_S^- = C_S - \{u_2\}$, *then if either of the following two conditions are met*

- $(u_1, u_2) \in E_i$ *and* $|N_{C_S^-}(u_1) \cap N_{C_S^-}(u_2)| < q - 2k - 2 \cdot \max\{k-2, 0\}$,
- $(u_1, u_2) \notin E_i$ *and* $|N_{C_S^-}(u_1) \cap N_{C_S^-}(u_2)| < q - k - \max\{k-2, 0\} - \max\{k-2, 1\}$,

then u_1 *and* u_2 *cannot co-occur in a* k*-plex* P *with* $|P| \geq q$.

Proof. Please see Appendix A.9 [3]. □

Theorem 5.15. *Let* v_i *be a seed vertex and* $G_i = (V_i, E_i)$ *be the corresponding seed subgraph. For any two vertices* $u_1, u_2 \in N_{G_i}(v_i) = C_S$, *let us define* $C_S^- = C_S - \{u_1, u_2\}$, *then if either of the following two conditions are met*

- $(u_1, u_2) \in E_i$ *and* $|N_{C_S^-}(u_1) \cap N_{C_S^-}(u_2)| < q - 3k$,
- $(u_1, u_2) \notin E_i$ *and* $|N_{C_S^-}(u_1) \cap N_{C_S^-}(u_2)| < q - k - 2 \cdot \max\{k-1, 1\}$,

then u_1 *and* u_2 *cannot co-occur in a* k*-plex* P *with* $|P| \geq q$.

Proof. Please see Appendix A.10 [3]. □

We next explain how Theorems 5.13, 5.14 and 5.15 are used in our algorithm to prune the search space. Recall that G_i is dense, so we use adjacency matrix to maintain the information of G_i. Here, we also maintain a boolean matrix T so that for any $u_1, u_2 \in V_i$, $T[u_1][u_2] = \mathit{false}$ if they are pruned by Theorem 5.13 or 5.14 or 5.15 due to the number of common neighbors in the candidate set being below the required threshold; otherwise, $T[u_1][u_2] = \mathit{true}$. Note that given T, we can obtain $T[u_1][u_2]$ in $O(1)$ time to determine if u_1 and u_2 can co-occur.

Recall from Figure 1 that we enumerate S via a set-enumeration tree. When we enumerate S in Algorithm 2 Line 7, assume that the current S is expanded from S' by adding u, and let $ext(S')$ be those candidate vertices that can still expand S', then by Theorem 5.13, we can incrementally prune those candidate vertices $u' \in ext(S')$ with $T[u][u'] = \mathit{false}$ to obtain $ext(S)$ that can expand S further.

We also utilize Theorem 5.14 to further shrink C_S in Algorithm 2 Line 8. Assume that the current S is expanded from S' by adding u, then we can incrementally prune those candidate vertices $u' \in C_{S'}$ with $T[u][u'] = \mathit{false}$ to obtain C_S.

Finally, we utilize Theorem 5.15 to further shrink C and X in Algorithm 3 Lines 2 and 3. Specifically, assume that v_p is newly added to P, then Line 2 now becomes

$$C \leftarrow \{v \in C \ \wedge \ P \cup \{v\} \text{ is a } k\text{-plex} \ \wedge \ T[v_p][v] = \mathit{true}\}.$$

Recall from Algorithm 2 Line 9 that vertices in X may come from V_i' or V_i. So for each $v \in X$, if $v \in V_i$, then we prune v if $T[v_p][v] = \mathit{false}$. This is also applied in Line 12 when we compute the new exclusive set to check maximality.

Variant of the Proposed Algorithm. Recall from Algorithm 3 Lines 15-16 that we always select the pivot v_p to be from C, so that our upper-bound-based pruning in Lines 17-18 can be applied to try to prune the branch in Line 19. In fact, if $v_p \in P$, FaPlexen [47] proposed another branching method to reduce the search space, also adopted by ListPlex [39]. Specifically, let us define $s = \sup_P(v_p)$, and $\overline{N_C}(v_p) = \{w_1, w_2, \ldots, w_\ell\}$, then we can move at most s vertices from $\overline{N_C}(v_p)$ to P to produce k-plexes. Note that $s \leq k$ since $\sup_P(v_p) = k - \overline{d_P}(v_p)$ so s is small, and that $s \leq \ell$ since otherwise, $P \cup C$ is a k-plex (because $s = \sup_P(v_p) > \ell = |\overline{N_C}(v_p)| = \overline{d_C}(v_p)$ means $k - (|P| - d_P(v_p)) > |C| - d_C(v_p)$, i.e., $d_{P \cup C} > |P| + |C| - k$) and this branch of search terminates (see Algorithm 3 Lines 11-14). Therefore, let the current task be $\langle P, C, X \rangle$, then it only needs to produce $s+1$ branches without missing k-plexes:

$$\langle P, \ C - \{w_1\}, \ X \cup \{w_1\} \rangle, \tag{4}$$

For $i = 2, \cdots, s$,

$$\langle P \cup \{w_1, \ldots, w_{i-1}\}, \ C - \{w_1, \ldots, w_i\}, \ X \cup \{w_i\} \rangle, \tag{5}$$

$$\langle P \cup \{w_1, \ldots, w_\ell\}, \ C - \{w_1, \ldots, w_\ell\}, \ X \rangle. \tag{6}$$

In summary, if $v_p \in P$, we can apply Eq (4)–Eq (6) for branching, while if $v_p \in C$, we can apply the upper bound defined by Eq (3) to allow the pruning of $\langle P \cup \{v_p\}, C - \{v_p\}, X \rangle$ in Algorithm 3 Line 19.

Therefore, besides the original Algorithm 3 denoted by **Ours**, we also consider a variant of Algorithm 3 which, when $v_p \in P$ is selected by Lines 7–10, uses Eq (4)–Eq (6) for branching rather than re-picking a pivot $v_p \in C$ as in Lines 15–16. We denote this variant by **Ours_P**. As we shall see in Section 7, Ours_P is generally not as time-efficient as Ours, showing that upper-bound based pruning is more effective than the branch reduction scheme of Eq (4)–Eq (6), so Ours is selected as our default algorithm in our experiments.

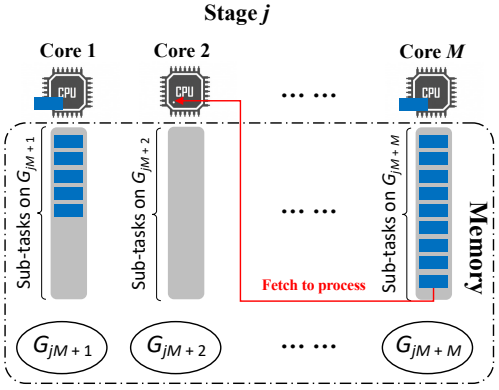


Figure 6: Illustration of Parallel Processing

6 PARALLELIZATION

Recall from Algorithm 2 that we generate initial task groups $T_{\{v_i\}}$ each creating and maintaining G_i. The sub-tasks of $T_{\{v_i\}}$ are $T_{\{v_i\} \cup S}$ that are generated by enumerating $S \subseteq N^2_{G_i}(v_i)$, and each such task runs the recursive $Branch(.)$ procedure of Algorithm 3 (recall Line 10 of Algorithm 2).

We parallelize Algorithm 2 on a multi-core machine with M CPU cores (and hence M working threads to process tasks) in stages. In each stage j ($j = 0, 1, \cdots$), the M working threads obtain M new task groups generated by the next M seed vertices in η for parallel processing, as illustrated in Figure 6.

Specifically, at the beginning of Stage j, the i^{th} thread creates and processes the task group with seed vertex v_{jM+i} by creating the subgraph G_{jM+i}, and enumerating S to create the sub-tasks with $P_S = \{v_{jM+i}\} \cup S$ and adding them into a task queue Q_i local to Thread i. Then, each thread i processes the tasks in its local queue Q_i to maximize data locality and hence cache hit rate (since the processing is on a shared graph G_i). If Thread i finishes all tasks in Q_i but other threads are still processing their tasks in Stage j, then Thread i will obtain tasks from another non-empty queue $Q_{i'}$ for processing to take over some works of Thread i'. This approach achieves load balancing while maximizing the CPU cache hit rate.

Stage j ends when the tasks in all queues are exhausted, after which we release the memory used by these task groups (e.g., seed subgraphs G_i) and move forward to Stage $(j+1)$ to process the next M seed vertices. The stages are repeated until all seed vertices in η are exhausted.

So far, we treat each sub-task T_{P_S} as an independent task run by a thread in its entirety. However, some sub-tasks T_{P_S} can become stragglers that take much longer time to complete than other tasks (e.g., due to a much larger set-enumeration subtree under P_S). We propose to use a timeout mechanism to further decompose each straggler task into many smaller tasks to allow parallel processing. Specifically, let t_0 be the time when the current task is created, and let t_{cur} be the current time. Then in Algorithm 3 Line 19 (resp. Line 20), we only recursively call $Branch(.)$ over $\langle P \cup \{v_p\}, C - \{v_p\}, X\rangle$ (resp. $\langle P, C - \{v_p\}, X \cup \{v_p\}\rangle$) if $t_{cur} - t_0 \leq \tau_{time}$, where τ_{time} is a user-defined task timeout threshold. Otherwise, let the thread processing the current task be Thread i, then we create a new task and add it to Q_i. The new tasks can reuse the seed subgraph of its task group, but need to materialize new status variables such as containers for keeping P, C and X, and the boolean matrix $T[u_1][u_2]$ for pruning by vertex pairs.

In this way, a straggler task will call $Branch(.)$ for recursive backtracking search as usual until $t_{cur} - t_0 \leq \tau_{time}$, after which it backtracks and creates new tasks to be added to Q_i. If a new task also times out, it will be further decomposed in a similar manner, so stragglers are effectively eliminated at the small cost of status variable materialization.

7 EXPERIMENT

In this section, we conduct comprehensive experiments to evaluate our method for large maximal k-plex enumeration, and compare it with the other existing methods. We also conduct an ablation study to show the effectiveness of our optimization techniques.

Datasets and Experiment Setting. Following [15, 16, 39], we use 18 real-world datasets in our experiments as summarized in Table 2, where n and m are the numbers of vertices and edges, respectively; Δ indicates the maximum degree and D is the degeneracy. These public graph datasets are obtained from Stanford Large Network Dataset Collection (SNAP) [2] and the Laboratory for Web Algorithmics (LAW) [1, 9]. Similar to the previous works [15, 16, 39], we roughly categorize these graphs into three types: small, medium, and large. The ranges of the number of vertices for these three types of graphs are $[1, 10^4)$, $[10^4, 5\times10^6)$, and $[5\times10^6, +\infty)$.

Our code is written in C++14 and compiled by g++-7.2.0 with the -O3 flag. All the experiments are conducted on a platform with 24 cores (Intel Xeon Gold 6248R CPU 3.00GHz) and 128GB RAM.

Existing Methods for Comparison. A few methods have been proposed for large maximal k-plex enumeration, including D2K [15], CommuPlex [47], ListPlex [39], and FP [16]. Among them, ListPlex [1] and FP [2] outperform all the earlier works in terms of running time according to their experiments [16, 39], so they are chosen as our baselines for comparison. Note that ListPlex and FP are concurrent works, so there is no existing comparison between them. Thus, we choose these two state-of-the-art algorithms as baselines and compare our algorithm with them. Please refer to Section 2 for a more detailed review of ListPlex and FP.

[1]https://github.com/joey001/ListPlex
[2]https://github.com/qq-dai/kplexEnum

Table 2: Datasets

Network	n	m	Δ	D
jazz	198	2742	100	29
wiki-vote	7115	100,762	1065	53
lastfm	7624	27,806	216	20
as-caida	26,475	53,381	2628	22
soc-epinions	75,879	405,740	3044	67
soc-slashdot	82,168	504,230	2552	55
email-euall	265,009	364,481	7636	37
com-dblp	317,080	1,049,866	343	113
amazon0505	410,236	2,439,437	2760	10
soc-pokec	1,632,803	22,301,964	14,854	47
as-skitter	1,696,415	11,095,298	35,455	111
enwiki-2021	6,253,897	136,494,843	232,410	178
arabic-2005	22,743,881	553,903,073	575,628	3247
uk-2005	39,454,463	783,027,125	1,776,858	588
it-2004	41,290,648	1,027,474,947	1,326,744	3224
webbase-2001	115,554,441	854,809,761	816,127	1506

In our description hereafter, efficiency means the running time unless otherwise specified. For all the tested algorithms, the running time includes the time for core decomposition, the time for subgraph construction, and the time for k-plex enumeration, but the graph loading time is excluded since it is a fixed constant.

Following the tradition of previous works, we set the k-plex size lower bound q to be at least $(2k-1)$ which guarantees the connectivity of the k-plexes outputted.

Parameter Setting. For experiments on sequential execution, we set parameters $k = 2, 3, 4$ and $q = 12, 20, 30$ following the parameter settings in [16, 39]. Note that some parameter settings return no valid maximal k-plexes, while others lead to existing algorithms running for prohibitive amount of time, so we avoid reporting those settings. For as-skitter, using a small q (e.g., 12, 20, 30) is too expensive so we use a larger value for q following [39].

For experiments on parallel execution, using experiments that can finish quickly cannot justify the need for parallel execution. We, therefore, pick the settings of q so that the job needs to run for some time to obtain quite some relatively large k-plexes.

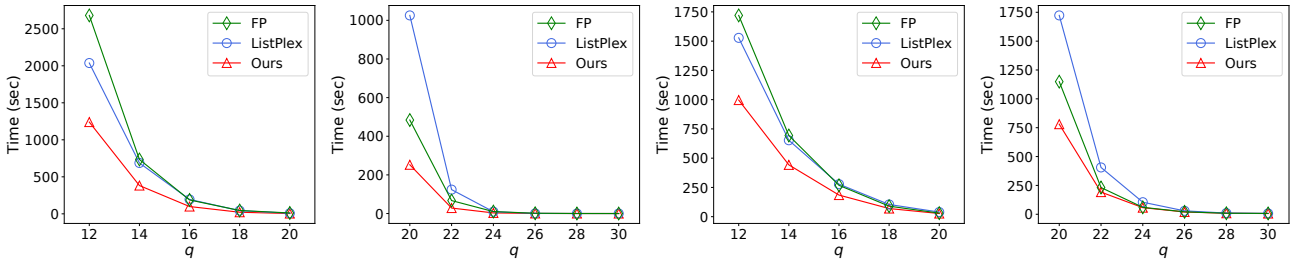
Performance of Sequential Execution. We first compare the sequential versions of our algorithm and the two baselines ListPlex and FP. Since sequential algorithms can be slow (esp. for the baselines), we use small and medium graphs.

We have extensively tested our algorithm by comparing its outputs with those of ListPlex and FP on various datasets, and have verified that all three algorithms return the same result set for each dataset and parameters (k, q). Table 3 shows the results where we can see that all three algorithms output the same number of k-plexes for each dataset and parameter (k, q). In terms of time-efficiency, our algorithm outperforms ListPlex and FP except for rare cases that are easy (i.e., where all algorithms finish very quickly). For our two algorithm variants, Ours is consistently faster than Ours_P, except for rare cases where both finish quickly in a similar amount of time. Moreover, our algorithm is consistently faster than ListPlex and FP. Specifically, Ours is up to 5× faster than ListPlex (e.g., soc-epinions, $k = 4$, $q = 30$), and up to 2× (e.g., email-euall, $k = 4$, $q = 12$) faster than FP, respectively. Also, there is no clear winner between ListPlex and FP: for example, ListPlex can be 3.56× slower than FP (e.g.,

Table 3: Running Time (sec) of Listing Large Maximal k-Plexes from Small and Medium Graphs by Various Algorithms

Network (n, m)	k	q	#k-plexes	Running time (sec)			
				FP	ListPlex	Ours_P	Ours
jazz (198, 2742)	4	12	2,745,953	3.68	4.12	3.92	**2.87**
lastfm (7624, 27,806)	4	12	1,827,337	2.39	2.58	2.52	**2.04**
as-caida (26,475, 53,381)	2	12	5336	**0.03**	**0.03**	**0.03**	**0.03**
	3	12	281,251	0.94	0.78	0.67	**0.53**
	4	12	15,939,891	51.39	45.31	37.52	**26.08**
amazon0505 (410,236, 2,439,437)	2	12	376	0.37	0.11	0.08	**0.07**
	3	12	6347	0.57	0.26	**0.20**	0.22
	4	12	105,649	1.47	0.99	0.91	**0.78**
as-skitter (1,696,415, 11,095,298)	2	60	87,767	**4.37**	5.14	4.41	4.68
		100	0	1.49	0.14	0.14	**0.11**
	3	60	9,898,234	283.52	1010.48	302.53	**234.17**
		100	0	1.49	0.14	0.14	**0.11**
email-euall (265,009, 364,481)	2	12	412,779	1.61	1.34	1.27	**1.11**
		20	0	0.08	**0.05**	**0.05**	**0.05**
	3	12	32,639,016	101.16	91.59	82.07	**56.22**
		20	2637	0.34	0.30	0.21	**0.19**
	4	12	1,940,182,978	7085.39	6041.73	5385.09	**3535.63**
		20	1,707,177	11.99	21.40	13.30	**7.70**
com-dblp (317,080, 1,049,866)	2	12	12,544	0.34	**0.11**	**0.11**	**0.11**
		20	5049	0.27	0.06	0.06	**0.05**
	3	12	3,003,588	6.13	3.75	3.68	**3.51**
		20	2,141,932	4.28	2.83	2.77	**2.57**
	4	12	610,150,817	914.84	729.16	720.13	**666.98**
		20	492,253,045	726.93	621.17	612.59	**546.30**

Network (n, m)	k	q	#k-plexes	Running time (sec)			
				FP	ListPlex	Ours_P	Ours
soc-epinions (75,879, 405,740)	2	12	49,823,056	278.56	153.64	157.98	**130.14**
		20	3,322,167	16.65	17.00	16.65	**14.01**
	3	20	548,634,119	2240.68	2837.49	2442.10	**1540.87**
		30	16,066	3.29	4.66	2.52	**2.11**
	4	30	13,172,906	139.59	545.82	198.88	**93.47**
wiki-vote (7115, 100,762)	2	12	2,919,931	19.21	14.77	13.62	**12.58**
		20	52	**0.37**	0.46	0.44	0.42
	3	12	458,153,397	2680.68	2037.51	1746.63	**1239.83**
		20	156,727	11.91	8.13	4.67	**4.15**
	4	20	46,729,532	483.97	1025.54	455.37	**252.40**
		30	0	**0.03**	0.07	0.08	0.06
soc-slashdot (82,168, 504,230)	2	12	30,281,571	91.10	65.57	66.16	**51.41**
		20	13,570,746	38.44	36.85	38.68	**28.89**
	3	12	3,306,582,222	10368.47	8910.28	8894.69	**5995.67**
		20	1,610,097,574	3950.16	5244.41	5325.80	**3377.68**
		30	4,626,307	32.36	76.44	49.43	**30.10**
	4	30	1,047,289,095	4167.85	10239.25	7556.32	**4016.08**
soc-pokec (1,632,803, 22,301,964)	2	12	7,679,906	50.77	39.16	36.78	**35.53**
		20	94,184	**9.02**	10.67	10.24	10.00
		30	3	5.98	5.83	5.72	**5.46**
	3	12	520,888,893	1719.85	1528.07	1347.97	**996.43**
		20	5,911,456	30.52	39.38	33.38	**26.94**
		30	5	6.10	6.35	6.47	**5.92**
	4	20	318,035,938	1148.87	1722.87	1292.65	**780.34**



(a) wiki-vote ($k = 3$) (b) wiki-vote ($k = 4$) (c) soc-pokec ($k = 3$) (d) soc-pokec ($k = 4$)

Figure 7: Running Time (sec) of the Three Algorithms on Various Datasets and Parameters

as-skitter, $k = 3$, $q = 60$), but over 10× faster in other cases (e.g., as-skitter, $k = 3$, $q = 100$).

We can see that our algorithm has more advantages in efficiency when the number of sub-tasks is larger. This is because our upper bounding and pruning techniques can effectively prune unfruitful sub-tasks. For instance, on dataset email-euall when $k = 4$ and $q = 12$, there are a lot of sub-tasks and the number of k-plexes outputted is thus huge; there, FP takes 7085.39 seconds while Ours takes only 3535.63 seconds.

We also study how the performance of sequential algorithms changes when q varies, and the results are shown in Figure 7. Due to space limitation, we only show results for four datasets, and more results can be found in Figure 14 in Appendix B.3 [3]. In each subfigure, the horizontal axis is q, and the vertical axis is the total running time. As Figure 7 shows, Ours (red line) consistently uses less time than ListPlex and FP. For example, Ours is 4× faster than ListPlex on wiki-vote when $k = 4$, $q = 20$.

As for the performance between ListPlex and FP, we can see from Figure 7 that when k is small, ListPlex (blue line) is always faster than FP (green line) with different values of q. As k becomes larger, FP can become faster than ListPlex. Note that the time complexity of ListPlex and FP are $O\left(n^{2k} + n(D\Delta)^{k+1}\gamma_k^D\right)$ and $O\left(n^2\gamma_k^n\right)$, respectively where $\gamma_k < 2$ is a constant. Therefore, when k is small, the time complexity of ListPlex is smaller than FP; but as k becomes large, the number of branches increases quickly and the upper-bounding technique in FP becomes effective. These results are also consistent with the statement in FP's paper [16]: *the speedup of FP increases dramatically with the increase of k*. As far as we know, this is the first time to compare the performance of ListPlex and FP, which are proposed in parallel very recently.

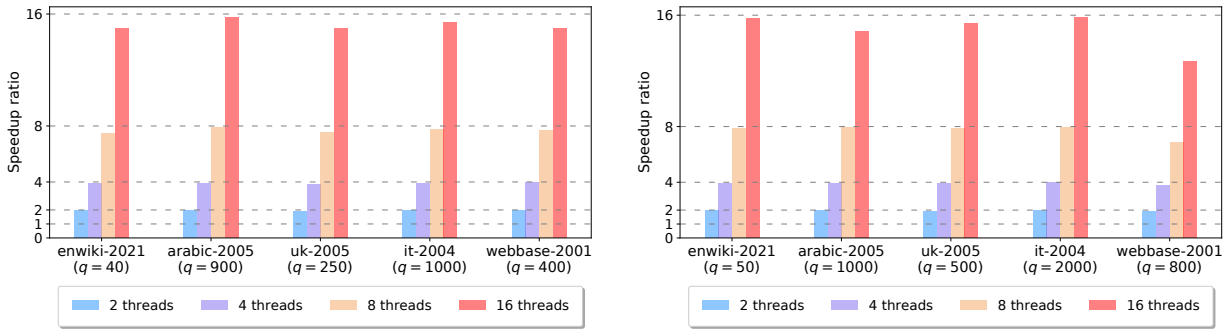
We notice that our reported running time of FP is slower than that reported in their paper [16] albeit the same settings, though our hardware is even more powerful. However, we double checked that we have faithfully run FP following their GitHub repo's instructions.

We also compare the peak memory consumption of three algorithms. Please see Appendix B.2 [3] for the results. To summarize, ListPlex and Ours report similar memory usage, while FP needs more memory to store the intermediate results even just on medium datasets. For example, the memory usage of FP, ListPlex, and Ours on soc-pokec are 937.52 MB, 431.69 MB, and 431.26 MB.

Performance of Parallel Execution. We next compare the performance of the parallel versions of Ours, ListPlex, and FP, using the large graphs. Note that both ListPlex and FP provide

Table 4: Running Time (sec) of Listing Large Maximal k-Plexes on Large Graphs by Parallel Algorithms (16 Threads)

Network	n	m	$k=2$							$k=3$						
			q	τ_{best} (ms)	#k-plexes	FP	ListPlex	Ours	Ours (τ_{best})	q	τ_{best}(ms)	#k-plexes	FP	ListPlex	Ours	Ours (τ_{best})
enwiki-2021	6,253,897	136,494,843	40	0.01	1,443,280	241.18	291.22	**154.99**	151.01	50	0.001	40,997	19056.73	3860.17	**1008.26**	1006.43
arabic-2005	2,2743,881	553,903,073	900	1	224,870,898	1873.71	417.91	**388.42**	385.52	1000	0.1	34,155,502	708.36	70.10	**67.98**	67.98
uk-2005	39,454,463	783,027,125	250	0.01	159,199,947	FAIL	194.68	**165.54**	164.20	500	0.1	116,684,615	553.03	56.68	**52.06**	52.06
it-2004	41,290,648	1,027,474,947	1000	20	66,067,542	1958.70	2053.83	**934.80**	907.36	2000	0.1	197,679,229	17785.82	458.83	**401.13**	401.13
webbase-2001	115,554,441	854,809,761	400	0.1	59674227	222.81	67.45	**60.93**	60.93	800	0.1	1,785,341,050	15446.46	3312.95	**3014.44**	3014.442



(a) $k = 2$ (b) $k = 3$

Figure 8: Speedup Ratio of Our Parallel Algorithm in Five Large Graphs

their own parallel implementations, but they cannot eliminate straggler tasks like Ours, which adopts a timeout mechanism (c.f. Section 6). For Ours, we fix the timeout threshold $\tau_{time} = 0.1$ ms by default to compare with parallel ListPlex and FP. We also include a variant "Ours (τ_{best})" which tunes τ_{time} to find its best value (i.e., τ_{best}) that minimizes the running time for each individual dataset and each parameter pair (k, q).

Table 4 shows the running time of parallel FP, ListPlex, Ours ($\tau_{time} = 0.1$ ms) and Ours ($\tau_{time} = \tau_{best}$) running with 16 threads. Note that the tuned values of τ_{best} are also shown in Table 4, which vary in different test cases. Please refer to Appendix B.1 [3] for the experimental results on tuning τ_{time}, where we can see that unfavorable values of τ_{time} (e.g., those too large for load balancing) may slow down the computation significantly. Overall, our default setting $\tau_{time} = 0.1$ ms consistently performs very close to the best setting when $\tau_{time} = \tau_{best}$ in all test cases shown in Table 4, so it is a good default choice.

Compared with parallel ListPlex and FP, parallel Ours is significantly faster. For example, Ours is 18.9× and 3.8× faster than FP and ListPlex on dataset enwiki-2021 ($k = 3$ and $q = 50$), respectively. Note that FP is said to have *very high parallel performance* [16] and ListPlex also claims that it *can reach a nearly perfect speedup* [39]. Also note that FP fails on uk-2005 when $k = 2$ and $q = 250$, likely due to a bug in the code implementation of parallel FP. In fact, FP can be a few times slower than ListPlex, since its parallel implementation does not parallelize the subgraph construction step: all subgraphs are constructed in serial at the beginning which can become the major performance bottleneck.

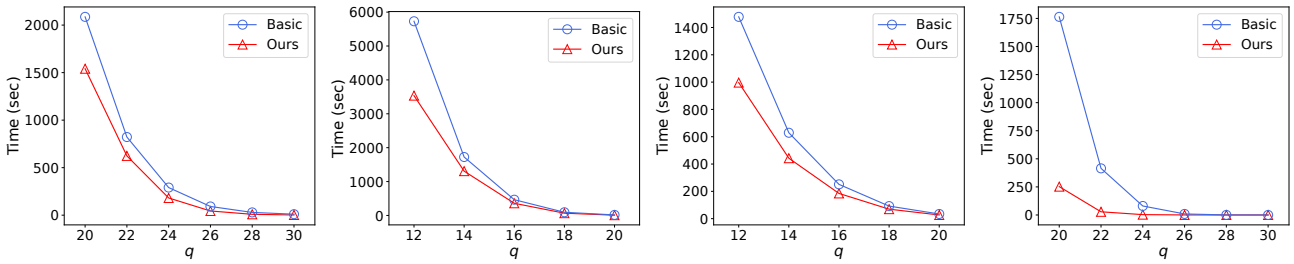
We also evaluate the scale-up performance of our parallel algorithm. Figure 8 shows the speedup results, where we can see that Ours scales nearly ideally with the number of threads on all the five large datasets for all the tested parameters k and q used in Table 4. For example, on dataset it-2004 ($k = 3$ and $q = 2000$), it achieves 7.93× and 15.82× speedup with 8 and 16 threads, respectively.

Table 5: Effect of Different Upper Bounding Techniques

Network	k	q	Running time (sec)		
			Ours\ub	Ours\ub+fp	Ours
wiki-vote	3	12	1393.50	1319.05	**1239.83**
		20	5.20	4.72	**4.15**
	4	20	530.48	280.75	**252.40**
		30	0.14	0.13	**0.06**
soc-epinions	2	12	138.82	142.06	**130.14**
		20	14.92	15.48	**14.01**
	3	20	1699.49	1687.29	**1540.87**
		30	2.87	2.44	**2.11**
email-euall	3	12	62.85	63.83	**56.22**
		20	0.29	0.28	**0.19**
	4	12	4367.88	3961.40	**3535.63**
		20	13.01	9.31	**7.70**
soc-pokec	3	12	1039.61	1022.14	**996.43**
		20	27.21	29.19	**26.94**
	4	20	988.90	877.95	**780.34**
		30	6.91	6.76	**6.37**

Ablation Study. We now conduct ablation study to verify the effectiveness of our upper-bound-based pruning technique as specified in Lines 17-18 of Algorithm 3, where the upper bound is computed with Eq (3). While ListPlex does not apply any upper-bound-based pruning, FP uses one that requires a time-consuming sorting procedure in the computation of upper bound (c.f., Lemma 5 of [16]).

The ablation study results are shown in Table 5, where we use "Ours\ub" to denote our algorithm variant without using upper-bound-based pruning, and use "Ours\ub+fp" to denote our algorithm that directly uses the upper bounding technique of FP [16] instead. In Table 5, we show the results on four representative datasets with different k and q (the results on other



(a) soc-epinions ($k = 3$) (b) email-euall ($k = 4$) (c) soc-pokec ($k = 3$) (d) wiki-vote ($k = 4$)

Figure 9: Running time (sec) of Our Basic and Optimized Algorithms on Various Datasets and Parameters

Table 6: Effect of Pruning Rules

Network	k	q	Running time (sec)			
			Basic	Basic+R1	Basic+R2	Ours
wiki-vote	3	12	2124.54	1726.29	1269.16	**1239.83**
		20	30.93	16.09	4.54	**4.15**
	4	20	1763.94	791.73	272.78	**252.40**
		30	0.14	0.15	0.15	**0.06**
soc-epinions	2	12	148.09	145.56	135.93	**130.14**
		20	16.42	16.31	14.86	**14.01**
	3	20	2086.60	1796.44	1582.69	**1540.87**
		30	10.48	6.43	2.30	**2.11**
email-euall	3	12	83.05	73.29	60.51	**56.22**
		20	0.53	0.44	0.28	**0.19**
	4	12	5729.98	4997.78	3708.91	**3535.63**
		20	15.39	11.58	8.04	**7.70**
soc-pokec	3	12	1477.74	1272.26	1009.03	**996.43**
		20	34.25	30.09	27.07	**26.94**
	4	20	1003.68	886.50	791.87	**780.34**
		30	6.88	6.57	6.63	**6.37**

datasets are similar and omitted due to space limit). We can see that Ours outperforms "Ours\ub" and "Ours\ub+fp" in all the cases. This shows that while using upper-bound-based pruning improves performance in this framework, FP's upper bounding technique is not as effective as Ours, due to the need of costly sorting when computing the upper bound in each recursion. In fact, "Ours\ub+fp" can be even slower than "Ours\ub" (c.f., soc-epinions with $k = 2$ and $q = 12$) since the expensive sorting procedure in the computation of upper bound backfires, while the upper-bound-based pruning does not reduce the branches much. Another observation is that our upper-bounding technique is more effective when k and the number of sub-tasks become larger (e.g., when q is smaller). For example, the running time of Ours and "Ours\ub" is 252.40 seconds and 530.48 seconds, respectively, on dataset wiki-vote with $k = 4$ and $q = 20$.

We next conduct ablation study to verify the effectiveness of our pruning rules, including (R1) Theorem 5.7 for pruning initial sub-tasks right before Line 10 of Algorithm 2, and (R2) Theorems 5.13, 5.14 and 5.15 for second-order-based pruning to shrink the candidate and exclusive sets during recursion.

The ablation study results are shown in Table 6, where our algorithm variant without R1 and R2 is denoted by "Basic". In Table 6, we can see that both R1 and R2 bring performance improvements on the four tested graphs. The pruning rules are the most effective on dataset wiki-vote with $k = 4$ and $q = 20$, where Ours achieves 7× speedup compared with Basic.

Figure 9 further compares the running time between Basic and Ours as k and q vary (where more values of q are tested). Due to space limitation, we only show results for four datasets, and more results can be found in Figure 15 in Appendix B.4 [3]. We can see that Ours is consistently faster than the basic version with different k and q. This demonstrates the effectiveness of our pruning rules.

8 CONCLUSION

In this paper, we proposed an efficient branch-and-bound algorithm to enumerate all maximal k-plexes with at least q vertices. Our algorithm adopts an effective search space partitioning approach that provides a good time complexity, a new pivot vertex selection method that reduces candidate size, an effective upper-bounding technique to prune useless branches, and three novel pruning techniques by vertex pairs. Our parallel algorithm version uses a timeout mechanism to eliminate straggler tasks. Extensive experiments show that our algorithms compare favorably with the state-of-the-art algorithms, and the performance of our parallel algorithm version scales nearly ideally with the number of threads.

ACKNOWLEDGMENTS

This work was supported by DOE ECRP Award 0000274975, NSF OIA-2229394, NSF OAC-2414474, and NSF OAC-2414185. Additionally, Zhongyi Huang was partially supported by the NSFC Project No. 12025104.

REFERENCES

[1] 2011. Laboratory for Web Algorithmics (LAW). https://law.di.unimi.it/datasets.php.
[2] 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.
[3] 2024. Full Technical Report. https://arxiv.org/abs/2402.13008v2.
[4] Yuichi Asahiro, Refael Hassin, and Kazuo Iwama. 2002. Complexity of finding dense subgraphs. *Discret. Appl. Math.* 121, 1-3 (2002), 15–26.
[5] Gary D Bader and Christopher WV Hogue. 2003. An Automated Method for Finding Molecular Complexes in Large Protein Interaction Networks. *BMC bioinformatics* 4, 1 (2003), 2.
[6] Balabhaskar Balasundaram, Sergiy Butenko, and Illya V. Hicks. 2011. Clique Relaxations in Social Network Analysis: The Maximum k-Plex Problem. *Oper. Res.* 59, 1 (2011), 133–142.
[7] Vladimir Batagelj and Matjaz Zaversnik. 2003. An O(m) Algorithm for Cores Decomposition of Networks. *CoRR* cs.DS/0310049 (2003). http://arxiv.org/abs/cs/0310049
[8] Devora Berlowitz, Sara Cohen, and Benny Kimelfeld. 2015. Efficient Enumeration of Maximal k-Plexes. In *SIGMOD*. ACM, 431–444.
[9] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *WWW*. ACM, 595–601.
[10] Coenraad Bron and Joep Kerbosch. 1973. Finding All Cliques of an Undirected Graph (Algorithm 457). *Commun. ACM* 16, 9 (1973), 575–576.

[11] Dongbo Bu, Yi Zhao, Lun Cai, Hong Xue, Xiaopeng Zhu, Hongchao Lu, Jingfen Zhang, Shiwei Sun, Lunjiang Ling, Nan Zhang, et al. 2003. Topological Structure Analysis of the Protein-Protein Interaction Network in Budding Yeast. *Nucleic acids research* 31, 9 (2003), 2443–2450.

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

[13] Moses Charikar. 2000. Greedy approximation algorithms for finding dense components in a graph. In *APPROX (Lecture Notes in Computer Science)*, Vol. 1913. Springer, 84–95.

[14] Alessio Conte, Donatella Firmani, Caterina Mordente, Maurizio Patrignani, and Riccardo Torlone. 2017. Fast Enumeration of Large k-Plexes. In *KDD*. ACM, 115–124.

[15] 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 *KDD*. ACM, 1272–1281.

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

[17] Uriel Feige, Guy Kortsarz, and David Peleg. 2001. The Dense k-Subgraph Problem. *Algorithmica* 29, 3 (2001), 410–421.

[18] Jian Gao, Jiejiang Chen, Minghao Yin, Rong Chen, and Yiyuan Wang. 2018. An Exact Algorithm for Maximum k-Plexes in Massive Graphs. In *IJCAI*. ijcai.org, 1449–1455.

[19] 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.

[20] Guimu Guo, Da Yan, Lyuheng Yuan, Jalal Khalil, Cheng Long, Zhe Jiang, and Yang Zhou. 2022. Maximal Directed Quasi-Clique Mining. In *ICDE*. IEEE, 1900–1913.

[21] John Hopcroft, Omar Khan, Brian Kulis, and Bart Selman. 2004. Tracking Evolving Communities in Large Linked Networks. *Proceedings of the National Academy of Sciences* 101, suppl 1 (2004), 5249–5253.

[22] Haiyan Hu, Xifeng Yan, Yu Huang, Jiawei Han, and Xianghong Jasmine Zhou. 2005. Mining Coherent Dense Subgraphs Across Massive Biological Networks for Functional Discovery. *Bioinformatics* 21, suppl_1 (2005), i213–i221.

[23] Daxin Jiang and Jian Pei. 2009. Mining frequent cross-graph quasi-cliques. *ACM Trans. Knowl. Discov. Data* 2, 4 (2009), 16:1–16:42.

[24] Hua Jiang, Fusheng Xu, Zhifei Zheng, Bowen Wang, and Wei Zhou. 2023. A Refined Upper Bound and Inprocessing for the Maximum k-Plex Problem. In *IJCAI*. ijcai.org, 5613–5621.

[25] 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 *IJCAI*. ijcai.org, 1689–1696.

[26] Jalal Khalil, Da Yan, Guimu Guo, and Lyuheng Yuan. 2022. Parallel mining of large maximal quasi-cliques. *VLDB J.* 31, 4 (2022), 649–674.

[27] 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.

[28] Junqiu Li, Xingyuan Wang, and Yaozu Cui. 2014. Uncovering the Overlapping Community Structure of Complex Networks by Maximal Cliques. *Physica A: Statistical Mechanics and its Applications* 415 (2014), 398–406.

[29] Guimei Liu and Limsoon Wong. 2008. Effective Pruning Techniques for Mining Quasi-Cliques. In *ECML/PKDD (Lecture Notes in Computer Science)*, Walter Daelemans, Bart Goethals, and Katharina Morik (Eds.), Vol. 5212. Springer, 33–49.

[30] Jeffrey Pattillo, Alexander Veremyev, Sergiy Butenko, and Vladimir Boginski. 2013. On the Maximum Quasi-clique Problem. *Discrete Applied Mathematics* 161, 1-2 (2013), 244–257.

[31] Jeffrey Pattillo, Nataly Youssef, and Sergiy Butenko. 2013. On Clique Relaxation Models in Network Analysis. *European Journal of Operational Research* 226, 1 (2013), 9–18.

[32] Jian Pei, Daxin Jiang, and Aidong Zhang. 2005. On mining cross-graph quasi-cliques. In *SIGKDD*. ACM, 228–238.

[33] Stephen B Seidman and Brian L Foster. 1978. A Graph-Theoretic Generalization of the Clique Concept. *Journal of Mathematical Sociology* 6, 1 (1978), 139–154.

[34] Steve Sheng, Brad Wardman, Gary Warner, Lorrie Cranor, Jason Hong, and Chengshan Zhang. 2009. An Empirical Analysis of Phishing Blacklists. In *6th Conference on Email and Anti-Spam (CEAS)*. Carnegie Mellon University, 1–10.

[35] Brian K. Tanner, Gary Warner, Henry Stern, and Scott Olechowski. 2010. Koobface: The Evolution of the Social Botnet. In *eCrime*. IEEE, 1–10.

[36] Charalampos E. Tsourakakis, Francesco Bonchi, Aristides Gionis, Francesco Gullo, and Maria A. Tsiarli. 2013. Denser than the densest subgraph: extracting optimal quasi-cliques with quality guarantees. In *KDD*. ACM, 104–112.

[37] Duygu Ucar, Sitaram Asur, Umit Catalyurek, and Srinivasan Parthasarathy. 2006. Improving Functional Modularity in Protein-Protein Interactions Graphs using Hub-Induced Subgraphs. In *European Conference on Principles of Data Mining and Knowledge Discovery*. Springer, 371–382.

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

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

[40] Chun Wei, Alan Sprague, Gary Warner, and Anthony Skjellum. 2008. Mining Spam Email to Identify Common Origins for Forensic Application. In *ACM Symposium on Applied Computing*, Roger L. Wainwright and Hisham Haddad (Eds.). ACM, 1433–1437.

[41] Daniel Weiss and Gary Warner. 2015. Tracking Criminals on Facebook: A Case Study from a Digital Forensics REU Program. In *Proceedings of Annual ADFSL Conference on Digital Forensics, Security and Law*. 1–11.

[42] Mingyu Xiao, Weibo Lin, Yuanshun Dai, and Yifeng Zeng. 2017. A Fast Algorithm to Compute Maximum k-Plexes in Social Network Analysis. In *AAAI*. AAAI Press, 919–925.

[43] Kaiqiang Yu, Cheng Long, Shengxin Liu, and Da Yan. 2022. Efficient Algorithms for Maximal k-Biplex Enumeration. In *SIGMOD*. ACM, 860–873.

[44] Zhiping Zeng, Jianyong Wang, Lizhu Zhou, and George Karypis. 2006. Coherent closed quasi-clique discovery from large dense graph databases. In *SIGKDD*. ACM, 797–802.

[45] Jiongzhi Zheng, Mingming Jin, Yan Jin, and Kun He. 2023. Relaxed Graph Color Bound for the Maximum k-plex Problem. *CoRR* abs/2301.07300 (2023).

[46] 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 *AAAI*. AAAI Press, 12453–12460.

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