ABSTRACT

Listing dense subgraphs in large graphs plays a key task in various network analysis applications like community detection. Clique, as the densest model, has been widely investigated. However, in practice, communities rarely form as cliques for various reasons, e.g., data noise. Therefore, $k$-plex – graph with each vertex adjacent to all but at most $k$ vertices, is introduced as a relaxed version of clique. Often, to better simulate cohesive communities, an emphasis is placed on connected $k$-plexes with small $k$. In this paper, we continue the research line of listing all maximal $k$-plexes and maximal $k$-plexes of prescribed size. Our first contribution is algorithm ListPlex that lists all maximal $k$-plexes in $O*(\gamma^D)$ time for each constant $k$, where $\gamma$ is a value related to $k$ but strictly smaller than 2, and $D$ is the degeneracy of the graph that is far less than the vertex number $n$ in real-word graphs. Compared to the trivial bound of $n^k$, the improvement is significant, and our bound is better than all previously known results. In practice, we further use several techniques to accelerate listing $k$-plexes of a given size, such as structural-based prune rules, cache-efficient data structures, and parallel techniques. All these together result in a very practical algorithm. Empirical results show that our approach outperforms the state-of-the-art solutions by up to orders of magnitude.

CCS CONCEPTS

- Information systems → Web mining; - Theory of computation → Graph algorithms analysis.

KEYWORDS

Listing maximal $k$-plexes, Graph algorithms, Worst-case time guarantee, Community detection, Parallelization

ACM Reference Format:
Zhengren Wang, Yi Zhou, Mingyu Xiao, and Bakhadyr Khoussainov. 2022. Listing Maximal $k$-Plexes in Large Real-World Graphs. In Proceedings of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

WWW '22, April 25–29, 2022, Virtual Event, Lyon, France
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9096-5/22/04...$15.00
https://doi.org/10.1145/3485447.3512198
worst-case [21]. Therefore, a large number of existing studies focus on the design of practically efficient methods. The majority of these algorithms have been derived and motivated by the Bron-Kerbosch algorithm [7], though it was originally designed to only list maximal cliques. Wu and Pei (2007) adapted the Bron-Kerbosch algorithm to list maximal \( k \)-plexes with a few new rules to prune unnecessary searches [27]. Wang et al. (2017) integrated more heuristic pruning rules and applied multi-thread parallelization technique [26]. Zhou et al. (2020) devised a novel branch heuristic with a worst-case time complexity proof. With their branch heuristic, the running time of Bron-Kerbosch algorithm is improved from \( O^*(2^n) \) to \( O^*(\gamma_k^n) \) where \( \gamma_k \) is related to \( k \) but strictly smaller than 2 \(^1\). Aside from the Bron-Kerbosch variants, there is another type of algorithms which have bounded delay between the output of two consecutive solutions. Berlowitz et al. (2020) initialized such kind of study by providing a polynomial-time delay algorithm for the problem [4, 13].

Listing large maximal \( k \)-plexes. We also study the problem of listing large maximal \( k \)-plexes, i.e., listing maximal \( k \)-plexes which have at least \( l \) vertices, \( l \) being a large number, say at least \( 2k - 1 \). The problem was originally proposed to amend two issues that arise in modeling the communities by maximal \( k \)-plexes [14–16, 31]. First, it is observed that there are enormous maximal \( k \)-plexes in real-world graphs, and empirically most of them are small or even unconnected. However, in community detection application, communities should be large and densely connected subgraphs. Second, existing maximal \( k \)-plex listing algorithms can only handle graphs with hundreds to thousands of vertices in days. But large sparse graphs are ubiquitous these days, e.g., the webbase-2001 graph has more than a hundred million vertices and more than a billion edges [6]).

Fortunately, by requiring that the output \( k \)-plexes must be at least larger than a threshold \( l (l \geq 2k - 1) \), the two issues can be alleviated. Due to the structural property of \( k \)-plexes (Property 3 in [28]), a \( k \)-plex with at least \( 2k - 1 \) vertices is densely connected, i.e., the shortest length of paths between every two vertices is not larger than 2. Therefore, the first issue does not exist. For the second issue, with a lower bound requirement on the size of output \( k \)-plexes, the performance of listing algorithm can be also accelerated with many powerful strategies [14–16]. For instance, in [15, 16], Conte et al. (2017) took advantage of the size constraint to remove a large portion of unfruitful vertices from the input graph, which made Berlowitz et al.’s listing algorithm possible to run on graphs of millions of vertices. Conte et al. (2018) further used decomposition and parallel techniques, leading to a listing algorithm capable of running on some web-scale graphs, e.g., the it-2004 graph. Zhou et al. (2020) used the same decomposition framework as in [14] so that their Bron-Kerbosch pivot heuristic can accommodate large real-world graphs.

1.2 Contributions
Motivated by the aforementioned studies, we develop the most efficient algorithm for listing both maximal \( k \)-plexes and large maximal \( k \)-plexes from sparse real-world graphs.

---

\(^1\)The notation \( O^* \) omits the polynomial factors.

1. We propose ListPlex, an algorithm that lists all maximal \( k \)-plexes with provably worst-case running time. The general idea of ListPlex is a marriage of new decomposition scheme and an efficient Bron-Kerbosch search. Our analysis discloses that for each constant \( k \), ListPlex has a worst-case time bound \( O^*(\gamma_k^n) \) where \( D \) is the degeneracy number of the input graph and \( \gamma_k \) is related to \( k \) but strictly smaller than 2. As far as we know, it is the first algorithm that reduces the exponent of running time from \( n \) to \( D \). Due to the power-law distribution of most real-world graphs, \( D \ll n \) in most cases, e.g., the webbase-2001 has more than a thousand million vertices but its degeneracy number is only 1506. To some extent, this bound provides theoretical evidence for the good performance of our algorithm.

2. We optimize the practical performance of ListPlex for listing large maximal \( k \)-plexes of size at least \( 2k - 1 \). It is known that listing large maximal \( k \)-plexes is of more real-world importance than purely listing all maximal \( k \)-plexes. Thus, we study efficient implementation techniques from multiple perspectives. From algorithmic perspective, we suggest strong prune rules to reduce the search space of our algorithm. From the computational system perspective, we propose new data structures to reduce cache misses and increase parallelism. All optimization techniques bring evident speedup for processing large sparse real-world graphs.

Our experiments show that ListPlex outperforms the state-of-the-art approaches in terms of both problems. For example, our parallel algorithm can list all large maximal \( 2 \)-plexes (with \( l = 800 \)) from the huge webbase-2001 graph with over one billion edges in 1 minute. This is almost an order of magnitude speedup compared to the best-known parallel approach.

All codes are available at https://github.com/joey001/ListPlex.git.

2 BACKGROUND
2.1 Basic notations
Let \( G = (V, E) \) be a simple and undirected input graph, where \( V \) and \( E \) are the sets of vertices and edges, respectively. We will let \( n = |V| \) and \( m = |E| \) in this paper. For \( v \in V \) and a positive integer \( k \), we use \( N_k^G(v) \) to denote the set of vertices with distance exactly \( k \) to \( v \) in \( G \). The vertices in \( N_k^G(v) \) are also called \( k \)-hop neighbors of \( v \). The set \( N_k^G(v) \) may be simply written as \( N_G(v) \) and 1-hop neighbors may be simply called neighbors. The degree of a vertex \( v \) is \( |N_G(v)| \). The maximum degree among all vertices in \( G \) is denoted by \( \Delta \). When the underlying graph \( G \) is clear from the content, we may ignore the subscript \( G \) and write \( N_k^G(v) \) as \( N_k^v \). Given a vertex set \( P \subseteq V \), let \( G[P] \) be the subgraph induced by \( P \). The diameter of \( G \) is the maximum distance among all pairs of vertices in \( G \).

A permutation of \( v_1 \ldots v_n \) is called a degeneracy ordering (or core ordering) of the graph \( G \) if for each \( i \), vertex \( v_i \) has the minimum degree in the induced subgraph \( G[\{v_1, \ldots, v_n\}] \). The degeneracy ordering of a graph can be computed in linear time by the algorithm that repeatedly removes a node with the minimum degree until the graph becomes empty [2]. For a degeneracy ordering \( \eta = v_1 \ldots v_n \), the degree of \( v_i \) in \( G[\{v_i, \ldots, v_n\}] \) is called the core number of \( v_i \). It is known that for any degeneracy ordering of the same graph, the largest core number among all vertices is the same and is called as degeneracy (or core number). We denote it by \( D \).
Let \( \eta \) be an ordering of \( V \), say \( \eta = v_1, \ldots, v_n \). For any two different vertices \( v_i \) and \( v_j \), denote \( v_i <_{\eta} v_j \) if \( v_i \) precedes \( v_j \) in \( \eta \), otherwise \( v_i >_{\eta} v_j \). For any \( v_i \), let \( N_{\eta>}(v_i) \) be \( N(v_i) \cap \{ v_1, \ldots, v_{i-1} \} \). \( N_{\eta<}(v_i) \) be \( N(v_i) \cap \{ v_{i+1}, \ldots, v_n \} \). Similarly, let \( N^2_{\eta<}(v_i) \) denotes \( N^2(v_i) \cap \{ v_1, \ldots, v_{i-1} \} \) and \( N^2_{\eta>}(v_i) \) denotes \( N^2(v_i) \cap \{ v_{i+1}, \ldots, v_n \} \).

As defined above, a \( k \)-plex is a graph such that each vertex is not adjacent to at most \( k \) vertices (including itself) in the graph. Thus, a 1-plex is a clique, i.e., a complete graph. A subgraph \( G' \) of \( G \) is called a maximal \( k \)-plex if \( G' \) is not a subgraph of any larger \( k \)-plex. So a maximal \( k \)-plex is always an induced subgraph. In this paper, we are interested in listing all maximal \( k \)-plexes of a graph.

**Problem 1** (Listing maximal \( k \)-plexes). Given a graph \( G = (V,E) \), a positive integer \( k \), list all maximal \( k \)-plexes of \( G \).

### 2.2 Some properties

We present basic properties of \( k \)-plexes. These are important for our algorithm design. Proofs of the lemmas below as well as missing proofs in the rest of the paper are left in the Appendix.

**Lemma 1.** Any induced subgraph of a \( k \)-plex is still a \( k \)-plex.

This property is known in the literature [13, 24, 30]. It will be frequently used in our algorithm. For example, we can validate the maximality of a \( k \)-plex, i.e., a \( k \)-plex \( G[P] \) is maximal if there is no vertex that can be added to \( G[P] \) such that \( G'[P] \) is still a \( k \)-plex.

**Lemma 2.** Any \( k \)-plex with at least \( 2k - 1 \) vertices is a connected graph with the diameter at most 2. A \( k \)-plex with at most \( 2k - 2 \) vertices may be disconnected.

Lemma 2 is also known in the literature [15, 29]. It shows \( 2k - 1 \) is a key boundary between connectedness and unconnectedness.

In practice, the \( k \)-plex is closely related to the community detection problem which asks for dense and large communities from a large network [14, 32]. Using maximal \( k \)-plex as a graph model of the community, we translate the community detection as listing maximal \( k \)-plexes that are at least connected, and with prescribed number of vertices. By Lemma 2, any \( k \)-plex of size at least \( 2k - 1 \) must be connected and even diameter-2 bounded. Therefore, it is rational to form the practical community detection as finding all maximal \( k \)-plexes of size at least \( l \), where \( l \) is a given lower bound value and \( l \) must be at least \( 2k - 1 \).

**Problem 2** (Listing large maximal \( k \)-plexes). Given a graph \( G = (V,E) \), two positive integers \( k \) and \( l \) where \( l \geq 2k - 1 \), list all maximal \( k \)-plexes with at least \( l \) vertices.

### 2.3 Existing Bron-Kerbosch based algorithms

Before we present our algorithm, we introduce the Bron-Kerbosch algorithm and its variants as they are closely related to ours.

**2.3.1 The fundamental Bron-Kerbosch Algorithm.** Many existing algorithms for listing maximal \( k \)-plexes, as in [3, 26, 27], stem from the Bron-Kerbosch algorithm that was originally designed for listing maximal cliques [7, 12]. We review the main idea of the Bron-Kerbosch algorithm for listing \( k \)-plexes.

The algorithm is recursive. We leave the pseudo-code in Alg. 2 in the Appendix. It calls a recursive procedure \( BKRec \) with three disjoint sets as parameters, i.e., \( P \), \( C \) and \( X \). \( P \) represents the set of vertices that should be contained in the \( k \)-plex in the current stage. \( C \) includes the remaining candidate vertices for enumerating. \( X \) contains excluded vertices. They are excluded from the \( k \)-plex to avoid non-maximal solutions.

\( BKRec \) lists all maximal \( k \)-plexes \( G[P'] \) satisfying the following three properties: (i) \( P \subseteq P' \), (ii) \( P' \subseteq P \cup C \), and (iii) \( \forall v \in X \), the subgraph \( G'[\{v\} \cup P] \) is not a \( k \)-plex.

Given a graph \( G = (V,E) \) and an integer \( k > 0 \), the algorithm calls \( BKRec \) initialized with \( P = X = \emptyset \) and \( C = V \). Then the algorithm iteratively branches on a vertex in \( C \) by including it to either \( P \) or \( X \). We will use \( BKPlex(G,k) \) to denote this algorithm.

**Complexity.** As mentioned in [31], the Bron-Kerbosch requires \( O^*(2^k) \) time in the worst-case, where \( n \) is the number of vertices in the input graph. Although several pruning rules were suggested for the Bron-Kerbosch in [26, 27], but the worst-case running time bound was not improved.

**2.3.2 Zhou et al.’s Pivot Heuristic.** Zhou et al. (2020) improved the Bron-Kerbosch algorithm with a pivot heuristic [31]. They observed that for any graph \( G \), either \( G \) is a \( k \)-plex or there is a vertex \( v \) not adjacent to at least \( k+1 \) vertices in \( G \), including itself. As such, they designed a pivot heuristic which always branches on the vertex of minimum degree in the graph.

**Complexity.** The pivot heuristic can reduce the total number of branches and then improve the worst-case running time from \( O^*(2^k) \) to \( O^*(\gamma_k^k) \), where \( \gamma_k \) is a number related to \( k \) but strictly smaller than 2.

**2.3.3 Conte et al.’s Decomposition Algorithm.** In [14], Conte et al. proposed a decomposition-based algorithm, namely D2K, for listing \( k \)-plexes with the diameter at most 2. D2K first sorts the vertices of \( G \) by degeneracy ordering \( v_1, \ldots, v_n \). Then, for each \( v_i \), D2K builds a subgraph \( G_i = (V_i, E_i) \) induced by \( \{ v_i \} \cup N_{\eta>}(v_i) \cup N^2_{\eta>}(v_i) \). The Bron-Kerbosch algorithm is then called to search all maximal \( k \)-plexes in \( G_i \). However, a maximal \( k \)-plex \( G'_i[P] \) of \( G_i \) is not a maximal \( k \)-plex of the original graph if a vertex preceding \( v_i \) can form a larger \( k \)-plex with \( P \). Hence, for every maximal \( k \)-plex \( G'_i[P] \) emitted by the Bron-Kerbosch search algorithm, D2K further validates that no other vertex in \( v_1, \ldots, v_{i-1} \) can form a \( k \)-plex with \( P \) before outputting it.

**Complexity.** D2K restricts the search space to each subgraph \( G_i = (V_i, E_i) \) and so the search size is bounded by \( O^*(\sum_{i}2^{|V_i|}) \). Recall that \( D \) is the degeneracy of the input graph \( G \) and \( \Delta \) is the maximum degree of \( G \). It holds that \( |V_i| \leq D\Delta \) for each \( i \). Thus, \( \sum_{i}2^{|V_i|} \leq n^2D\Delta \). Due to the sparsity of many real-world graphs, \( \Delta \) and \( D \) are normally small values. The algorithm thus performs better than the Bron-Kerbosch algorithm in these large graphs.

### 3 LISTING ALL MAXIMAL \( k \)-PLEXES

We present our algorithm, ListPlex, for listing all maximal \( k \)-plexes.

**3.1 The main Structure**

Our algorithm contains two parts that are to list maximal \( k \)-plexes of size at most \( 2k - 2 \) vertices and at least \( 2k - 1 \) vertices, respectively. As mentioned in Lemma 2, maximal \( k \)-plexes of size at most \( 2k - 2 \)
may not be connected and this kind of k-plex is not interesting in practice. In fact, usually the parameter k is also small and most previous algorithms only tested the cases of k ≤ 5. In our algorithm, we will modify the Bron-Kerbosch algorithm by adding the size constraint to find all maximal k-plexes of size at most 2k − 2.

Next, we will focus on listing maximal k-plexes of size at least 2k − 1. By Lemma 2, we know that maximal k-plexes of size at least 2k − 1 are connected graphs with the diameter at most 2. So following the idea of Conte et al.’s decomposition algorithm, we list maximal k-plexes containing a vertex νi by only considering the local subgraph induced by {νi} ∪ Nνi(νi) ∪ N2νi(νi). However, we further use some techniques to reduce the search space again and get a significantly improved running time bound.

### 3.2 Listing maximal k-plexes larger than 2k − 2

In this subsection, we focus on listing all maximal k-plexes of size at least 2k − 1. The pseudo-code corresponds to the second part in Alg. 1. We will explain the idea and each step of the algorithm.

First, ListPlex sorts the V by a degeneracy ordering η = ν1, ..., νn. From ν1 to νn, ListPlex iteratively lists maximal νi-led k-plexes with at least 2k − 1 vertices.

**Definition 1.** Given an ordering η = ν1, ..., νn of the vertices of G, a νi-led k-plex is a k-plex G[S] such that νi is in S and it holds that νi ≻ η u for each vertex u ∈ S \ {νi}. A νi-led k-plex is maximal if it is not a subgraph of any k-plex in G[{ν1, ..., νn}].

Note that a maximal νi-led k-plex may not be maximal in the original graph G. So in our algorithm, when a maximal νi-led k-plex is found, we also check its maximality in G.

The core part of the algorithm is to find all maximal νi-led k-plexes. Instead of using a brute force method, we dramatically reduce the search space by utilizing the structural properties.

**Lemma 3.** Given an ordering η of the vertices of G, let G[P] be a νi-led k-plex induced by P and |P| ≥ 2k − 1. Then G[P] must be a subgraph of G[{νi} ∪ Nνi(νi) ∪ N2νi(ηi)}]. Furthermore, P contains at most k − 1 vertices from N2νi(ηi).

Let us call Gi = G[{νi} ∪ Nνi(νi) ∪ N2νi(ηi)}] the seed graph of νi. Given a νi and a subset S ⊆ N2νi(νi) such that |S| ≤ k − 1, let us call Ps = {νj} ∪ S as a seed set. For set Ps = {νj} ∪ S by Lemma 3, we call BKpivot to search maximal νi-led k-plexes that must contain Ps. The elaboration of BKpivot is left to the next subsection. In the current stage, we specify that for each seed set Ps = {νj} ∪ S, BKpivot emits all maximal k-plexes that must include Ps, possibly include some vertices in Nνi(ηi) and must not include vertices in N2νi(νi) \ S.

For each maximal νi-led k-plex G[P] found by BKpivot, ListPlex further tests the maximality of G[P] in the input graph G. That is to say, if a vertex in Nνi(νi) and N2νi(νi) can form a larger k-plex with P, then G[P] is not maximal in G. Otherwise, G[P] is maximal and P is emitted.

### 3.3 The BKpivot algorithm

We introduce BKpivot. It is also a branching algorithm following the style of the basic Bron-Kerbosch algorithm and it accepts three disjoint sets P, C and X playing the same roles as those in the Bron-Kerbosch algorithm. However, it additionally integrates some ideas into its branch scheme to reduce more vertices.

The pseudo-code is given in Alg. 3 in the Appendix. The recursive procedure, BKpivot(G, k, P, C, X), lists all maximal k-plexes that must subsume P, possibly include vertices in C and must not contain any vertex in X. The idea relies on the fact that, if G[P ∪ C] is a k-plex, then no further branches will be produced. Otherwise, there is a vertex in P ∪ C that has at least k + 1 non-neighbors in G[P ∪ C], including itself. In detail, BKpivot first checks the maximality of P. Afterwards, a vertex up of minimum degree in G[P ∪ C] is selected as pivot and BKpivot branches as follows:

- If up is not adjacent to at most k vertices in P ∪ C, then G[P ∪ C] is a k-plex. In this case, we check if G[P ∪ C] is maximal in G. If so, emit P ∪ C and stop the current branch.
- Otherwise, upk is not adjacent to q vertices in P ∪ C, where q ≥ k + 1. The consecutive branches are generated with respect to either upk ∈ P and upk ∈ P.
- If upk ∉ P, we generate two branches by either moving upk from C to X or moving upk from P to P. The latter case will fall into the next case.
- If upk ∈ P, let |P \ N(upk)| = q1 and |C \ N(upk)| = q2. Then q1 + q2 = q. It is not hard to prove q1 < k and let k′ = k − q1. Thus, at most k′ vertices in C \ N(upk) can be included in the k-plex. Denote C \ N(upk) as [u1, ..., uq2] by an arbitrary order. We generate k′ + 1 branches:
  1. In the first branch, u1 is moved from C to X;
  2. In the second branch, u1 is moved from C to P and u2 is moved from C to X;
  3. In the ith branch where i is from 3 to k′, [u1, ..., ui−1] are moved from C to P, and ui is moved from C to X;
  4. In the last branch, [u1, ..., uk′] are moved from C to P and [uk′+1, ..., uk2] are moved from C to X.
Correctness relies on Steps (a)-(d). Each maximal $k$-plex will fall into one case of (a)-(d). In the last case (d), the maximal $k$-plexes that include $\{u_1, \ldots, u_{\eta}\}$ are visited. Because $u_p$, and $\{u_1, \ldots, u_{\eta}\}$ are in $P$, so $\{u_{2\eta+1}, \ldots, u_{2\eta}\}$ can be excluded from further consideration since at most $k'$ vertices in $C \setminus N(u_p)$ can be included in the $k$-plex. Fig. 2 shows an example of the branch scheme.

![Diagram](image)

**Figure 2**: An example of BKPivot’s branch scheme with $k = 4$. In node 1, pivot $u_p = 1 \in P, P \setminus N(u_p) = \{1, 2\}$ and $C \setminus N(u_p) = \{3, 4, 5, 7\}$. At most $k' = k - |P \setminus N(u_p)| = 2$ non-neighbors of $u_p$ can be moved from $C$ to $P$. Node 1 generates three branches, i.e., node 2, node 3 and node 4. In node 3, there are several vertices of minimum degree. Assume pivot $u_p = 7 \in C$, node 3 further generates two branches by moving $u_p$ to $X$ or $P$, i.e., node 5 and node 6.

### 3.4 Complexity analysis

The main complexity result is below. See Appendix for the proof.

**Theorem 1.** Given a graph $G = (V, E)$ with maximum degree $\Delta$ and degeneracy $D$, ListPlex($G, k$) lists all maximal $k$-plexes without repetition in time $O(n^{2k} + n(D\Delta)^{k+1}k^{-k})$, where $y_k < 2$ is the largest root of $1 = x^{-1} + \cdots + x^{-k-1}$.

**Remark.** Note that for each $k$, the exponential part of the running time of our algorithm is $y_k^D$ and $y_k$ is bounded by $O(2 - \frac{1}{7D})$. The exponential part for the Bron-Kerbosch algorithm is $2^n$. The exponential part for Conte et al.’s decomposition algorithm is $2^{DA}$, the exponential part for Zhou et al.’s algorithm is $y_k^D$. Hence, our algorithm provides a significant improvement of the previously known state-of-the-art algorithms. By keeping the status of at most $k$ vertices at each branch, the BKPivot also greatly optimizes the space complexity of Zhou et al.’s pivot heuristic.

### 4 LISTING LARGE MAXIMAL $k$-PLEXES

In order to list large maximal $k$-plexes, i.e., maximal $k$-plexes of size at least $l (l \geq 2k - 1)$, ListPlex can be reused by simply prohibiting the output of $k$-plexes smaller than $l$. However, this mildly changed algorithm is previewed to be inefficient in practice. In fact, it is possible to prune some branches early and improve the practical performance due to the import of this size constraint. For example, because $l \geq 2k - 1$, the search for maximal $k$-plexes of size at most $2k - 2$ (Part I of Alg 1) can be simply dropped. For more stronger pruning techniques, let us first introduce an important observation.

**Lemma 4.** Assume $G[P]$ is a $k$-plex of $G = (V, E), |P| \geq l$. For any two vertices $u, v \in P$, if $(u, v) \in E$, then $|N(u) \cap N(v) \cap P| \geq l - 2k$, otherwise $|N(u) \cap N(v) \cap P| \geq l - 2k + 2$.

Note that this property was also observed in [14, 31].

#### 4.1 Pruning seed graph $G_i$

Suppose the degeneracy ordering of $G = (V, E)$ is $\eta = v_1, \ldots, v_n$. Recall that when we search the maximal $v_i$-led $k$-plexes, we build a seed graph $G_i$ which is an induced graph of $\{v_1\} \cup N(v_1) \cup N^{2}_{v_1}(v_2)$. Denote the vertex and edge sets of $G_i$ are $V_i$ and $E_i$, respectively. We show rules to reduce the scale of $G_i$.

**Prune Rule 1.** Assume $u \in V_i$, if $u$ satisfies
\[ u \in N_{P, \delta} (v_i) \text{ and } |N(u) \cap N(v_i) \cap V_i| < 1 - 2k, \]
\[ \text{or } u \in N_{P, \eta} (v_i) \text{ and } |N(u) \cap N(v_i) \cap V_i| < 1 - 2k + 2, \]

then \( u \) can be excluded from \( G_i \) without affecting the correctness of ListPlex.

### 4.2 Excluding unfruitful seed sets

Intuitively, if we can identify some unfruitful seed sets \( P_k \), i.e., sets that are impossible to be a part of large \( k \)-plexes, we can save the forthcoming exponential search in \( G_i \). With this in mind, we make use of the following pruning rule.

**Prune Rule 2.** Given a seed graph \( G_i = (V_i, E_i) \), a seed set \( P_k = \{ v_i \} \cup S \) where \( S \subseteq N_{P, \eta} (v_i) \) and \( |S| \leq k - 1 \). Denote \( C_\delta = N_{P, \delta} (v_i) \). For any two vertices \( u, v \in S \), if
- \( (u, v) \in E \) and \( |N_{G_i}(u) \cap N_{G_i}(v) \cap C_\delta| < 1 - 2k - \max(k - 3, 0) \),
- \( \text{or } (u, v) \not\in E \text{ and } |N_{G_i}(u) \cap N_{G_i}(v) \cap C_\delta| < 1 - 2k + 2 - \max(k - 3, 0) \),

then \( P_k \) is not in any maximal \( v_i \)-led \( k \)-plexes of size at least \( l \).

It turns out that this prune rule dramatically improves the performance of our algorithm. In Fig. 3, we show the comparison between the algorithm using Prune Rule 2 and the one without it.

![Figure 3: The number of seed sets and running time with and without Prune Rule 2.](image)

### 5 IMPLEMENTATION TECHNIQUES

We present important techniques to implement ListPlex on modern computers: computers with multi-level caches and multiple cores.

#### 5.1 Reducing cache misses

In an initial implementation, the algorithm searches maximal \( k \)-plexes by visiting \( G_i \) and \( G \) alternatively. When a maximal \( v_i \)-led \( k \)-plex \( G[P] \) is found from \( G_i \), ListPlex revisits the input graph \( G \) to validate if a vertex in \( v_1, \ldots, v_{l-1} \) forms a larger \( k \)-plex with that solution. This results in a high amount of cache misses when checking the maximality of \( G[P] \). Clearly, it is partially caused by the fact that the data of \( G \) is swapped out from the cache.

In order to reduce cache misses, we further make use of the diameter-2 property of large \( k \)-plexes. For each vertex \( v_i \) in ordering \( \eta \), we build a bipartite graph \( B_i = (L_i, R_i, F_i) \) where \( L_i = N_{\leq \eta} (v_i) \cup N_{\eta}^2 (v_i), R_i = \{ v_i \} \cup N_{\leq \eta} (v_i) \cup N_{\eta}^2 (v_i) \) and edge set \( F_i \subseteq L_i \times R_i \) is induced from \( G \). When BK Pivot finds a maximal \( v_i \)-led \( k \)-plex \( G[P] \) on \( G_i \), we further validate if for each vertex \( u \in L_i \),
- \( |N_{B_i}(u) \cap P| \leq |P| + 1 - k \)
- \( \exists \emptyset \in P \text{ that } |N_{G_i}(\emptyset) \cap P| = |P| - k \text{ and } (u, \emptyset) \not\in E \).

then \( G[P] \) is maximal in \( G \). With \( B_i \), to check the maximality of a \( k \)-plex, we only need to visit \( G_i \) and \( B_i \). Though the vertex numbers of \( G_i \) and \( B_i \) are both \( O(DA) \), in real-world graphs, the vertex numbers of \( G_i \) and \( B_i \) are far less than \( |V| \), implying good locality.

We compare the time and cache misses between the algorithm using \( B_i \) and the one without \( B_i \) in Figure 4.

![Figure 4: The total number of data cache misses and the running time with and without using bipartite graph \( B_i \).](image)

### 5.2 Parallelization

ListPlex also owns appealing parallel features. We introduce a shared-memory parallel version of ListPlex in this subsection.

It is observed that searches of maximal \( v_i \)-led \( k \)-plexes are independent for each \( v_i \). Thus, for each vertex \( v_i \), we create a task, say \( T_i \), to process the search of all maximal \( v_i \)-led \( k \)-plexes. \( T_i \) owns its private seed graph \( G_i \) and bipartite graph \( B_i \). Tasks \( T_1, \ldots, T_n \) can be executed in parallel.

However, it could happen that most tasks stop but a few heavy tasks are still running. Specifically, when the number of running tasks is less than the number of available cores, computational resources are wasted. In such a case, we split the branches of a running task \( T_i \) into new subtasks for the idle cores. Assume that \( T_n \) has been dispatched for execution but \( T_i \) (\( i < n \)) is still staying in the BK Pivot procedure. Then, \( T_i \) detects some idle cores, it spawns recursive calls to BK Pivot for \( (G_i, k, P, C, X) \) as subtasks of \( T_i \) and dispatches them to idle cores. A subtask of \( T_i \) owns its sets \( P, C \) and \( X \) but shares \( G_i \) and \( B_i \) with \( T_i \). Indeed, the schedule follows the work-stealing scheduling algorithm which accommodates well with the Bron-Kerbosch algorithm [5].

**Fine-Grained Task.** In parallel computing, the granularity of subtasks substantially affects the performance. Empirically, simple small tasks should not be spawned due to the overhead. In our implementation, we measure the complexity of a subtask, i.e., the time of executing BK Pivot for \( (G_i, k, P, C, X) \), by the size of \( C \). Particularly, if \( |C| > 10 \) and there are some idle cores, we spawn new subtasks and assign them to available cores.
6 EXPERIMENTS

Experiments setup. The codes are written in C++11 and compiled by g++-9.3.0 with optimization option `-O3'. All experiments are conducted on a computer with a Ubuntu20.04 operating system, two-way Intel Xeon Gold 6130 CPUs (2.1GHz, 22MB L3-cache, 2 CPU chips and 32 physical cores in total), a 132G RAM and a 1T SSD. We also disable hyper-threading and turbo techniques. ListPlex is parallelized with the OpenMP library.

Dataset. In Table 1, we report basic information of benchmark graphs, including the number of vertices $n$, number of undirected edges $m$, maximum degree $\Delta$ and degeneracy $D$. These graphs are taken from Stanford Large Network Dataset Collection (SNAP) [20] and Laboratory for Web Algorithms (LAW) [2]. As we can see, the size of these graphs broadly ranges. Like [14], we divide them into three categories, i.e., small, medium and large graphs. Large graphs have more than ten million nodes, medium graphs are those with more than ten thousand nodes while the remaining graphs are classified as small graphs.

Table 1: Considered networks and their properties

| Network | $n$  | $m$  | $\Delta$ | $D$  |
|---------|------|------|----------|------|
| jazz    | 198  | 2742 | 100      | 29   |
| cs-grqc | 5241 | 14484| 81       | 43   |
| gnutella08 | 3601 | 41554| 97       | 10   |
| wiki-vote | 7116 | 100763| 1065    | 53   |
| fastlm  | 7624 | 55612| 216      | 20   |
| as-caida | 26475| 53381| 2628     | 22   |
| soc-epinions | 75888| 405739| 3044     | 67   |
| soc-slashdot | 82144| 500480| 2548     | 54   |
| email-euall | 265214| 365659| 7636     | 37   |
| amazon0505 | 410236| 2439436| 2760    | 10   |
| in-2004  | 1353703| 13126172| 21869   | 488  |
| soc-pokec | 1632803| 22301964| 14854   | 47   |
| as-skitter | 1696415| 11095298| 35455   | 111  |
| soc-livejournal | 4847571| 68993773| 14415   | 360  |
| arabic-2005 | 22744080| 639999458| 575628  | 3247 |
| uk-2005  | 39459925| 936364282| 1372171| 584   |
| it-2004  | 41291594| 1150725436| 1243927| 3209  |
| webbase-2001 | 118142155| 1019903190| 816127  | 1506  |

6.1 Listing all maximal $k$-plexes

In this section, we evaluate the performance of our ListPlex for listing all maximal $k$-plexes. We compare our ListPlex with the fastest known algorithm BKPivot [31] and the traditional Bron-Kerbosch algorithm BKPlex. Note that the competitive D2K [14] solver only outputs large maximal $k$-plexes, i.e., $k$-plexes of size at least $l$ where $l > 2k - 2$. The recent solvers GP [26] and Enum [4] are not as time-efficient as BKPivot, see [31]. In case a solver cannot finish in 12 hours (43200 seconds) for an instance, we imperatively stop it. In the table, we mark the unfinished instances with OOT.

In Table 2, we show the time performance of these listing algorithms. We also report the parallel running time of ListPlex with 16 threads and the parallel speedup. Due to the huge amount of maximal $k$-plexes, neither of these algorithms is able to list all of them on medium or large graphs in 12 hours, even setting $k = 2$.

In terms of time, ListPlex outperforms both competitors for all these instances. For cases like wiki-vote with $k = 2$, ListPlex runs like 7× faster than the other algorithms. ListPlex also achieves a nearly perfect speedup for almost all cases except very simple ones. Unexpectedly, BKPlex runs faster than BKPivot for the last two larger graphs when $k = 2$.

Table 2: Listing all maximal $k$-plexes in small graphs

| Network       | $k$ | $k$-plexes | The running time (s) | Speedup |
|---------------|-----|------------|----------------------|---------|
|               |     |            | BKPlex | BKPivot | ListPlex | ListPlex[16] |           |
| jazz          | 2   | 35214      | 648.864 | 0.29    | 0.086    | 0.408       | 0.211     |
| jazz          | 3   | 3602575    | 772.826 | 17.55   | 6.477    | 0.832       | 7.785     |
| jazz          | 4   | 199305833  | 3226.746| 829.40  | 417.646  | 26.187      | 15.949    |
| gnutella08    | 2   | 1371419    | 1085.02 | 649.985 | 40.880   | 2.660       | 13.899    |
| gnutella08    | 3   | 1986959    | 1500.208| 3627.57 | 1117.858 | 70.207      | 13.922    |
| wiki-vote     | 2   | 46193264   | 2036.533| 30176.72| 1526.884 | 95.856      | 13.962    |
| fastlm        | 2   | 29886853   | 2643.394| 8667.89 | 1989.701 | 124.525     | 11.758    |

6.2 Listing large maximal $k$-plexes

We evaluate the problem of listing large maximal $k$-plexes, i.e., maximal $k$-plexes that have at least $l$ vertices. There are a rich number of solvers, e.g., GP [26], LP [15], D2K [14] and CommuPlex [31] for the problem. According to their empirical results, D2K and CommuPlex outperform earlier GP and LP in terms of practical running time. Thus, we compare our ListPlex with D2K and CommuPlex in this subsection. Also, D2K only outputs diameter-2 bounded maximal $k$-plexes. By setting $l$ at least $2k - 1$, we make sure that three compared algorithms output the same set of $k$-plexes. Also, we set a cut-off time of 12 hours for each instance.

The Sequential Performance. Let us first compare the sequential versions of D2K, CommuPlex and ListPlex. In Table 3, we show the sequential running time of different algorithms. For small networks, we set $k = 2, 3$ and 4, and $l = 12, 20$ and 30. For medium networks, we also set $k = 2, 3$ and 4 but we change $l$ for different graphs, mainly because all three algorithms cannot list all the 2 to 4-plexes even $k = 30$ in 12 hours. As for the large networks, we leave the test in the parallel environment. These large graphs contain a dramatic number of maximal $k$-plexes that cannot be efficiently listed by these sequential algorithms.

ListPlex is the best performing algorithm for these instances. Exceptions can only be observed in graphs which contain very few maximal $k$-plexes, e.g., wiki-vote with $k = 2$ and $l = 30$. For the rest of these instances, ListPlex achieves a 4-100× speedup over CommuPlex and a 3-420× speedup over D2K. For example, ListPlex is able to list all 4-plexes with $l = 20$ for wiki-vote in half an hour but CommuPlex and D2K cannot finish in 12 hours. For some instances like soc-slashdot with $k = 4$ and $l = 30$, ListPlex is the only algorithm that lists all maximal $k$-plexes of size at least $l$. It is worth observing that, the running time of D2K and CommuPlex contrasts in different scenarios, e.g., D2K runs 10× faster than CommuPlex in in-2004 with $k = 2$ but CommuPlex performs much better in soc-epinions with $k = 2$ or 3. In total, the results show the great superiority of ListPlex over the existing algorithms.

The Parallel Performance. It is known that D2K also provides a parallel version that achieves almost linear speedup for many
Table 3: The running time of listing large maximal k-plexes from small and medium graphs by CommuPlex, D2K and ListPlex.

| Graph | I | k-plexes | The running time (s) |
|-------|---|---------|---------------------|
| arabic-2005 | 2 | 800 | 242670893 | CommuPlex | 2195.272 |
| arabic-2005 | 2 | 1000 | 2568972 | D2K | 2284.435 |
| arabic-2005 | 2 | 1200 | 34155502 | ListPlex | 345 |
| uk-2005 | 2 | 250 | 108248475 | CommuPlex | 975.769 |
| uk-2005 | 2 | 500 | 258806 | D2K | 1239.717 |
| uk-2005 | 2 | 1000 | 34155502 | ListPlex | 345 |
| uk-2005 | 2 | 1200 | 242670893 | CommuPlex | 2195.272 |
| uk-2005 | 2 | 1400 | 2568972 | D2K | 2284.435 |
| uk-2005 | 2 | 1600 | 34155502 | ListPlex | 345 |
| web-base-2001 | 2 | 800 | 15497045 | CommuPlex | 236.141 |
| web-base-2001 | 2 | 1000 | 15497045 | D2K | 236.141 |
| web-base-2001 | 2 | 1200 | 15497045 | ListPlex | 345 |

Table 4: The parallel running time of large networks by ListPlex and D2K with 16 threads.

| Graph | I | k-plexes | The running time (s) |
|-------|---|---------|---------------------|
| arabic-2005 | 2 | 800 | 242670893 | CommuPlex | 2195.272 |
| arabic-2005 | 2 | 1000 | 2568972 | D2K | 2284.435 |
| arabic-2005 | 2 | 1200 | 34155502 | ListPlex | 345 |
| uk-2005 | 2 | 250 | 108248475 | CommuPlex | 975.769 |
| uk-2005 | 2 | 500 | 258806 | D2K | 1239.717 |
| uk-2005 | 2 | 1000 | 34155502 | ListPlex | 345 |
| uk-2005 | 2 | 1200 | 242670893 | CommuPlex | 2195.272 |
| uk-2005 | 2 | 1400 | 2568972 | D2K | 2284.435 |
| uk-2005 | 2 | 1600 | 34155502 | ListPlex | 345 |
| web-base-2001 | 2 | 800 | 15497045 | CommuPlex | 236.141 |
| web-base-2001 | 2 | 1000 | 15497045 | D2K | 236.141 |
| web-base-2001 | 2 | 1200 | 15497045 | ListPlex | 345 |

7 CONCLUSION

We studied the problems of listing maximal k-plexes and maximal k-plexes of prescribed size. We proposed ListPlex, a fast and scalable algorithm that efficiently solves the two problems in real-world graphs. Especially, ListPlex combines a new decomposition scheme with the branching algorithm, achieving a better theoretical complexity. When maximal k-plexes of size at least l (l ≥ 2k − 1) are asked, ListPlex can be also used for listing these large maximal k-plexes. For practical considerations, we designed some additional prune rules for listing large maximal k-plexes. These prune rules work very well in the context of large real-world graphs. Furthermore, we designed a new local bipartite graph to improve the cache performance of the algorithm, and parallel scheduling strategies to increase parallelism. Extensive empirical evaluations show the superiority of ListPlex over the state-of-the-art approaches for both problems.

ACKNOWLEDGMENTS

This work is supported by National Natural Science Foundation of China under grant nos. 61802049, 61972070 and 62172077.
A  THE BRON–KERBOBCH ALGORITHM

Algorithm 2: The Basic Bron-Kerbosch Algorithm

1. BKPlex(G, k)
2. begin
3. BKRec(G, k, ∅, V, ∅)
4. BKRec(G, k, P, C, X)
5. begin
6. \( C \leftarrow \{ v \in C : G[P \cup \{ v \}] \text{ is a } k\text{-plex} \} \)
7. \( X \leftarrow \{ v \in X : G[P \cup \{ v \}] \text{ is a } k\text{-plex} \} \)
8. if \( C = \emptyset \) then
9. \( \text{ if } X = \emptyset \text{ then return} \)
10. \( \text{ emit } P \)
11. \( \text{ return} \)
12. else
13. \( \text{ for } u \in C \) do
14. \( C \leftarrow C \setminus \{ u \} \)
15. BKRec(G, k, P \cup \{ u \}, C, X)
16. \( X \leftarrow X \cup \{ u \} \)
17. end

B  THE BKPIVOT ALGORITHM

Algorithm 3: The Bron-Kerbosch algorithm with pivot heuristic for listing all maximal \( k \)-plexes.

1. BKPivot(G, k, P, C, X)
2. begin
3. \( C \leftarrow \{ v : v \in C \text{ and } G[\{ v \} \cup P] \text{ is a } k\text{-plex} \} \)
4. \( X \leftarrow \{ v : v \in X \text{ and } G[\{ v \} \cup P] \text{ is a } k\text{-plex} \} \)
5. if \( C = \emptyset \) then
6. \( \text{ if } X = \emptyset \text{ then return} \)
7. \( \text{ emit } P \)
8. \( \text{ return} \)
9. \( \text{ Find a vertex of minimum degree } u_P \text{ in } G[P \cup C] \)
10. \( \text{ if } |N(u_P)| \geq |P| + |C| - k \) then
11. \( \text{ if } \exists v \in X \text{ that } G[\{ P \cup C \cup \{ v \} \}] \text{ is a } k\text{-plex} \) then
12. \( \text{ emit } P \cup C \)
13. else if \( u_P \in P \) then
14. \( \text{ Let } u_1, \ldots, u_q \text{ be an arbitrary ordering of } C \setminus N(u_P) \)
15. \( \text{ for } i \in \{ 2, \ldots, k' \} \text{ do} \)
16. \( \text{ BKPivot}(G, k, P, \{ u_1, \ldots, u_{i-1} \}, X \cup \{ u_i \}) \)
17. \( \text{ BKPivot}(G, k, P, \{ u_1, \ldots, u_{i-1}, u_i \}, C \setminus \{ u_i \}) \)
18. \( \text{ BKPivot}(G, k, P \cup \{ u_1, \ldots, u_q \}, C \setminus \{ u_1, \ldots, u_q \}, X) \)
19. \( \text{ BKPivot}(G, k, P \cup \{ u_1, \ldots, u_q \}, X \cup \{ u_P \}) \)
20. \( \text{ BKPivot}(G, k, P, C \setminus \{ u_P \}, X \cup \{ u_P \}) \)
22. \( \text{ BKPivot}(G, k, P, C \setminus \{ u_P \}, X) \)

C  MISSING PROOFS

Proof of Lemma 2

Proof. Let \( u \) and \( v \) be any pair of nonadjacent vertices in a \( k \)-plex. There are at most \( k - 1 \) vertices not adjacent to \( u \) and at most \( k - 1 \) vertices not adjacent to \( v \). If the graph has more than \( 2k - 2 \) vertices, then there exists a vertex \( w \) that is adjacent to both of \( u \) and \( v \). So the graph is connected and the diameter is at most \( 2 \).

Here is an example of a disconnected \( k \)-plex of size \( 2k - 2 \). The graph consists of two cliques of size \( k - 1 \). We can see that the graph is a \( k \)-plex since each vertex is not adjacent to \( k \) vertices (including itself). The number of vertices in the graph is \( 2k - 2 \). □

Proof of Lemma 3.

Proof. By Lemma 2, we know that the diameter of \( G[P] \) is bounded by \( 2 \). Since \( G[P] \) contains \( u_1 \), we know that \( G[P] \) can only be a subgraph of \( G[\{ v_1 \} \cup N_{G}(v_1) \cup N_{G}^2(v_1)] \). The second claim holds due to the definition of \( k \)-plexes. □

Proof of Lemma 4.

Proof. Let us denote \( O = P \setminus \{ u, v \} \). Then \( |O| \geq q - 2 \).

- If \( u \) and \( v \) are adjacent, there are at most \( 2(k - 1) \) vertices that are not common neighbors of \( u \) and \( v \) in \( O \). Thus, \( |N(u) \cap N(v) \cap P| \leq |O| - 2(k - 1) \leq (l - 2) - (2k - 2) = l - 2k \).
- If \( u \) and \( v \) are not adjacent, then there are at most \( 2(k - 2) \) nonneighbors in \( O \). Thus \( |N(u) \cap N(v) \cap P| \leq |O| - 2(k - 2) \leq (l - 2) - (2k - 4) = l - 2k + 2 \). □

Proof of Theorem 1.

Proof. For the first part, the running time is bounded by the number of subsets of size at most \( 2k - 2 \) times the time to check its maximality. There are at most \( O((\frac{n}{2k-2})) = O(n^{2k-2}) \) subsets of size at most \( 2k - 2 \). By [31], the time to check the maximality of a \( k \)-plex is bounded by \( O(n^2) \). So the running time is \( O(n^2k) \).

Before analyzing the part for listing maximal \( k \)-plexes of size at least \( 2k - 1 \), we first consider the running time bound of the procedure BKPivot. When \( C = \emptyset \), we do not need to branch anymore. So we analyze our branching operations by measuring the number of vertices removed from \( C \). The branching operation for the case \( u_P \in P \) will generate \( k' + 1 \) subbranches. In the first subbranch, one vertex \( u_1 \) is removed from \( C \). In the second subbranch, two vertices \( u_1, u_2 \) are removed from \( C \). In the ith branch for \( 3 \leq i \leq k' \), exactly \( i \) vertices \( \{ u_1, \ldots, u_i \} \) are removed from \( C \). In the last branch, \( q_2 \) vertices \( \{ u_1, \ldots, u_q \} \) are removed from \( C \). Where \( q_2 \geq k' + 2 \). If we use \( T(c) \) to denote the running time of BKPivot working on \( C \) with \( c = |C| \), then we get the following recurrence

\[
T(c) \leq T(c - 1) + \cdots + T(c - k') + T(c - q_2).
\]

When \( u_P \not\in P \ (u_P \in C) \), we generate two branches each of which will remove one vertex \( u_P \) from \( C \). In the latter case, we will follow with the above recurrence. Combining them together, we have

\[
T(c) \leq T(c - 1) + \cdots + T(c - k' - 1) + T(c - q_2 - 1).
\]
Note that $k' \leq k - 1$ and $q_2 \geq k' + 1$. For the worst case that $k' = k - 1$ and $q_2 = k' + 1$, we get the recurrence

$$T(c) \leq T(c - 1) + \cdots + T(c - k) + T(c - k - 1).$$

Let $y_k$ be the largest root of function $1 = x^{-1} + \cdots + x^{-k-1}$. Then the running time bound of the algorithm is bounded by $O(y_k^{-1} |C|)$. In our algorithm, initially $C$ is $N(v_i)$ and then $|C| \leq D$, where $D$ is the degeneracy of the graph. We also note that $y_k$ is strictly smaller than 2. For example, when $k = 1, 2, 3, 4$ and 5, $y_k = 1.618, 1.839, 1.928, 1.966$ and 1.984, respectively. Details on solving recurrence relations and time analysis can be found in [19].

Next, we analyze the algorithm for listing maximal $k$-plexes of size at least $2k - 1$. Note that computing the degeneracy order of a graph $G$ is in $O(m)$ [2]. For each vertex $v_i$ in the degeneracy order, we find all maximal $v_i$-led $k$-plexes in the subgraph $G_i = G \{v_i\} \cup N_{\leq q}(v_i) \cup N_{\geq q}(v_i)$). Hereby, we enumerate all subsets $S \subseteq N_{\geq q}(v_i)$ with size $|S| \leq k - 1$ and for each $S$ we include it to $P$ to generate an instance. So we will generate at most $|N_{\geq q}(v_i)|^k$ instances. For each instance, we will call BKPivot with running time $O(y_k^{-1} |N_{\geq q}(v_i)|)$. Additionally, in order to validate the maximality of a maximal $v_i$-led $k$-plex in $G$, the algorithm tries if any vertex in $N_{\geq q}(v_i) \cup N_{\geq q}(v_i)$ can form a $k$-plex with $P$. So, this will at most add a factor of $|N_{\geq q}(v_i)| + |N_{\geq q}(v_i)| \leq D + D \Delta$. In total, the running time is in $O((D + D \Delta) \sum |N_{\geq q}(v_i)| y_k^{-1}|N(v_i)|) = O(n(D \Delta)^{k+1} y_k^{-1}).$

**Proof of Prune Rule 1.**

**Proof.** Fix $v$ with the leading vertex $v_i$ in Lemma 4.

- If $u \in N_{> q}(v_i)$, then $(u, v_i) \in E$. Thus for any vertex $u \in P$,
  $$|N(u) \cap N(v_i) \cap V_i| \geq |N(u) \cap N(v_i) \cap P| \geq 1 - 2k,$$
  - If $u \in N_{\geq q}(v_i)$, then $(u, v_i) \notin E$. Thus for any vertex $u \in P$,
  $$|N(u) \cap N(v_i) \cap V_i| \geq |N(u) \cap N(v_i) \cap P| \geq k - 2 + 2.$$

**Proof of Prune Rule 2.**

**Proof.** It is clear $|N_{G_1}(u) \cap N_{G_2}(v) \cap \{v_i\}| + |N_{G_1}(u) \cap N_{G_2}(v) \cap S| + |N_{G_1}(u) \cap N_{G_2}(v) \cap C_i| \geq |N(u) \cap N(v) \cap P|$. Because $u, v \in S$, then $|N_{G_1}(u) \cap N_{G_2}(v) \cap \{v_i\}| = 0$ and $|N_{G_1}(u) \cap N_{G_2}(v) \cap S| \leq k - 1 - 2 = k - 3$. Thus, $|N_{G_1}(u) \cap N_{G_2}(v) \cap \{v_i\}| + |N(u) \cap N(v) \cap P| - \max(k - 3, 0)$. According to Lemma 4, $|N(u) \cap N(v) \cap P|$ has a lower bound depending on whether $(u, v) \in E$ or not. Combining that, we present Prune Rule 2 as above.