Counting Cohesive Subgraphs with Hereditary Properties

Rong-Hua Li
Beijing Institute of Technology
Beijing, China
lironghuabit@126.com

Xiaowei Ye
Beijing Institute of Technology
Beijing, China
yexiaowei@bit.edu.cn

Fusheng Jin
Beijing Institute of Technology
Beijing, China
jfs21cn@bit.edu.cn

Yu-Ping Wang
Beijing Institute of Technology
Beijing, China
wyp_cs@bit.edu.cn

Ye Yuan
Beijing Institute of Technology
Beijing, China
yuan-ye@bit.edu.cn

Guoren Wang
Beijing Institute of Technology
Beijing, China
wanggrbit@gmail.com

arXiv:2405.04823v1 [cs.DS] 8 May 2024

ABSTRACT

The clique model has properties of hereditariness and cohesiveness. Here hereditary property means a subgraph of a clique is still a clique. Counting small cliques in a graph is a fundamental operation of numerous applications. However, the clique model are often too restrictive for practical use, leading to the focus on other relaxed-cliques with properties of hereditariness and cohesiveness. To address this issue, we investigate a new problem of counting general hereditary cohesive subgraphs (HCS). All subgraphs with properties of hereditariness and cohesiveness can be called a kind of HCS. To count HCS, we propose a general framework called HCSPivot, which can be applied to count all kinds of HCS. HCSPivot can count most HCS in a combinatorial manner without explicitly listing them. Two additional noteworthy features of HCSPivot is its ability to (1) simultaneously count HCSs of any size and (2) simultaneously count HCSs for each vertex or each edge. The implementation of HCSPivot for clique counting is exactly the state-of-the-art clique counting algorithm PIVOTER [33]. We focus specifically on two HCS: s-defective clique and s-plex. We also propose several non-trivial pruning techniques to enhance the efficiency. We conduct extensive experiments on 8 large real-world graphs, and the results demonstrate the high efficiency and effectiveness of our solutions.

1 INTRODUCTION

Counting small cohesive subgraphs in a graph is a fundamental problem in graph mining and has a wide range of applications in various fields, such as community detection, network analysis, and bioinformatics [18, 51]. Among different types of motifs, clique is considered as one of the most important motifs due to the perfect cohesiveness and hereditary properties. Cohesiveness means great reach-ability between the vertices and hereditary property means that any induced subgraph of a clique is still a clique. In real-world, a group often has properties of cohesiveness and hereditary. For example, a group of friends should have abilities to reach each other and any subgroups are still friends. Therefore, counting cliques serves as a basic operator in many applications [7, 40, 53, 59, 60, 71].

However, the constraint of clique is often very strict for real-world applications because the interaction between members of a group may not be direct, or there may be missing data in a real-world system. To overcome this limitation, many relaxed clique models have been proposed. Notable examples include s-defective clique [14, 20, 72], s-plex [19, 21, 57, 75], s-clique [41], γ-quasi-clique [10, 49], k-core [56], k-truss [31, 64], and k-edge connected subgraph [13, 74]. In this work, we focus mainly on the s-defective clique and s-plex models, since these two models are often close to a clique and also they maintain both the cohesiveness and hereditary properties.

Specifically, a graph is an s-defective clique if there exist at most s missing edges compared to cliques. Since the maximum count of missing edges is restricted, s-defective clique is typically very cohesive. The subgraphs of an s-defective clique are still s-defective cliques because the deletion of nodes will not lead to the increasing of missing edges. Different from s-defective clique, an s-plex restricts each vertex has at most s missing edges, i.e., has at most s non-neighbors. Similar to s-defective clique, s-plex also has the property of hereditary because the deletion of nodes will not lead to increasing the count of non-neighbors of any vertex.

Instead of counting cliques in a graph, we study a new and more general problem of counting hereditary cohesive subgraphs (HCSs in short) in a graph. To address this problem, we develop two general counting frameworks which can be applied to counting both s-defective cliques and s-plexes. Similar to clique counts, the counts of HCSs have diverse applications in graph analysis. Below, we highlight two specific applications.

Motif-based Graph Clustering. Motif-based graph clustering has been recognized as the state-of-the-art method for detecting real-world communities in a network [7, 60]. The motif-based graph clustering method aims to minimize the so-called motif-based conductance, which is an important concept in network analysis whose definition is mainly based on the count of motifs [7, 60]. Hence, the key for motif-based graph clustering methods is to compute the count of motifs. Our experiments show that compared to using traditional clique counts [7, 60], using the count of HCSs, such as s-defective clique and s-plex, can improve the quality of clustering. This highlights the practical importance of the problem of counting hereditary cohesive subgraphs.

Network comparison. Comparing the counts of hereditary cohesive subgraphs across different graphs can aid in network comparison and similarity analysis. By quantifying the presence and abundance of these subgraphs, we can measure the structural similarities or differences between graphs. This enables us to identify graph properties or patterns that are shared or distinct, contributing to comparative network analysis and classification tasks. In this work, we define a new graph profile metric based on the count of HCSs. Our experiments show that such a new metric outperforms existing metrics in distinguishing different types of networks, suggesting that the counts of HCSs offer great potential as a novel structural feature for analyzing complex networks.

Challenges. Although counting HCSs has many practical applications, it is a challenging task. First, HCSs have a more complex structure than cliques (the cliques are a subset of HCSs), and the counts of HCSs are often much larger than that of cliques, making HCSs harder to enumerate and count. For example, on the Epinion network, the count of 8-cliques is 4.53×10^8 while the count of 1-defective clique with size 8 is 4.02×10^9 and the count of 1-plex

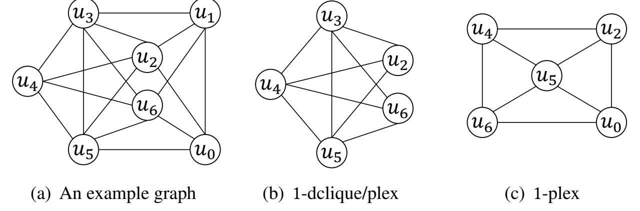
with size 8 is 1.95×10^{10}. Second, there is no previous work for counting s-defective clique and s-plex. Although there are several algorithms for counting cliques, they are not suitable for counting HCSs. This is because the complex structures of HCSs requires new search strategies, pruning techniques and optimizations that are not present in the clique counting algorithms. As a result, new algorithms and techniques need to be developed to tackle the challenges of counting HCSs.

Contributions. To overcome the computation challenges of counting HCSs, we first propose a listing-based backtracking framework, called HCSList. This framework lists all HCSs only once, and thus it can also obtain the count. HCSList utilizes the hereditary property to grow HCSs from small to large through backtracking. Based on the HCSList framework, we devise two specific algorithms to count s-defective cliques and s-plexes respectively. To improve the efficiency of these algorithms, we also develop a k-core based pruning technique and an upper bounding technique which can significantly reduce the unpromising vertices and unnecessary search branches respectively.

The main limitation of HCSList is that it needs to list all HCSs which is often intractable for counting relatively-large (e.g., size > 10) HCSs in large graphs. To overcome this issue, we propose a novel pivot-based framework, called HCSPivot, which can count most HCSs in a combinatorial manner without explicitly listing them. Instead of listing all HCSs with a given size q, HCSPivot lists *large* HCSs (not necessarily maximal) and then counts HCSs with size q in each large HCSs using a combinatorial method based on the hereditary property. To avoid repeatedly counting, HCSPivot makes use of a carefully-designed pivoting technique which can uniquely represent each HCS using the large HCSs. Since listing the *large* HCSs is much cheaper than listing all HCSs with size q, HCSPivot is often tractable to handle large graphs. Moreover, two important and useful features of HCSPivot are that (1) it can simultaneously count HCSs of any size, and (2) it can also obtain *local count* of HCSs for each vertex or each edge, where the local count of a vertex (or an edge) means the number of HCSs containing that vertex (or edge). Based on the HCSPivot framework, we also develop two specific algorithms with two novel pivot-vertex selection strategies for counting s-defective cliques and s-plexes respectively. The results of comprehensive experiments demonstrate the high efficiency and effectiveness of our solutions. In summary, the main contributions of this work are as follows.

Novel HCS *counting frameworks.* We propose two novel frameworks to count HCSs in a graph, namely HCSList and HCSPivot respectively. HCSList counts all HCSs by exhaustively enumerating them which is efficient when the number of HCSs is not very large. HCSPivot, however, counts most HCSs in a combinatorial manner without listing them, thus it is typically much more efficient than HCSList when the number of HCSs is large. Both of these two frameworks are very general, and they can be easily applied to count any hereditary cohesive subgraph in a graph. To our knowledge, we are the first to study the HCS counting problem and provide systematic and efficient approaches to solve this problem.

New algorithms for s-defective clique and s-plex counting. We first propose two counting algorithms for the s-defective clique and s-plex counting problems based on the proposed HCSList framework. We also present several non-trivial pruning techniques to boost the efficiency of these algorithms. Then, based on the HCSPivot framework, we develop another two counting algorithms with two carefully-designed pivot-vertex selection strategies for counting s-defective cliques and s-plexes respectively.



(a) An example graph (b) 1-dclique/plex (c) 1-plex

Figure 1: Running example.

Extensive experiments. We conduct comprehensive experiments using 8 real-world large graphs to evaluate the efficiency of our algorithms. The results show that (1) both HCSList and HCSPivot are efficient when counting very small HCSs, and HCSPivot is several orders of magnitude faster than HCSList for counting relatively-large HCSs. For example, on DBLP (425957 vertices and 2099732 edges), when $s = 1$ and $q = 9$ (the given size), HCSList takes 162301.4 seconds and 193757.0 seconds to count all s-defective cliques and s-plexes respectively. However, with the same parameters, HCSPivot consumes only 0.1 seconds and 0.2 seconds to count all s-defective cliques and s-plexes respectively, which is around 7 orders of magnitude faster than HCSList. (2) For both locally counting in all vertices (or edges) and simultaneously counting HCSs with size q in a given range, the time costs of HCSPivot are within the same order of magnitude as those for computing the total count of HCSs in a graph with a given size q. These results further demonstrate the high efficiency of our pivot-based solutions.

We also perform extensive experiments to evaluate the effectiveness of our algorithms through two applications, including motif-based graph clustering and graph profile based network characterization. The results demonstrate that (1) the HCS (including both s-defective clique and s-plex) count is very useful for motif-based graph clustering applications, which is often more effective than the clique counts, and it can also achieve the state-of-the-art performance for real-world community detection. (2) The HCS based graph profile is very effective to characterize different types of networks, while the state-of-the-art clique based method is often failed to distinguish various kinds of networks.

For reproducibility purpose, the source code of our work is available at [39].

2 PRELIMINARIES

Denote by $G = (V, E)$ an undirected graph where V is the set of vertices and $E \subseteq V \times V$ is the set of edges. The neighbors of each vertex $u \in V$ is $N(u) \triangleq \{v | (u, v) \in E\}$. The 2-hop neighbors of each vertex u is $N_2(u) \triangleq \{w | (u, v) \in E, (w, v) \in E, (u, w) \notin E\}$. Given a graph $G(V, E)$ and a vertices set $Q \subseteq V$, define the edges induced by Q as $E(Q) \triangleq \{(u, v) | (u, v) \in E, u \in Q, v \in Q\}$. We use $G(Q) = (Q, E(Q))$ to denote the subgraph induced by Q. For representation simplicity, we replace the size $|Q|$ with q. We define the total missing edges of $G(Q)$ as $\overline{m}(Q) = \binom{q}{2} - |E(Q)|$, and define the missing edges of a specific vertex u as $\overline{m}(u, Q) = |Q \setminus \{u\}| - |N(u) \cap Q|$. Note that $|Q \setminus \{u\}| = q - 1$ if $u \in Q$, $|Q \setminus \{u\}| = q$ otherwise.

A k-core of G is a maximal subgraph in which every vertex has a degree no less than k within the subgraph [56]. The core number of a vertex u denotes the maximum k such that there is a k-core containing u. The degeneracy ordering of V is an ordering $\{v_1, v_2, ...\}$ such that v_i has the minimum degree in the subgraph $G(\{v_i, v_{i+1}, ...\})$ [45]. A nice property of degeneracy ordering is that $\forall i, |N(v_i) \cap \{v_{i+1}, v_{i+2}, ..\}| \leq \delta$, where δ is the value of degeneracy. Note that the degeneracy value δ is equal to the maximum core

number of the vertices in G, which is often very small in real-world networks [12, 25]. In the ordered graph, we define the (2-hop) out-going neighbors of a vertex v_i as $\vec{N}(v_i) = N(v_i) \cap \{v_{i+1}, v_{i+2}, ...\}$ and $\vec{N}_2(v_i) = N_2(v_i) \cap \{v_{i+1}, v_{i+2}, ...\}$.

Definition 2.1 (Hereditary graph). A graph with property $\mathcal{P}$ is hereditary if all induced subgraphs also meet $\mathcal{P}$.

Definition 2.1 gives the concept of hereditary graphs. Among the hereditary subgraphs, we are interested in *Hereditary Cohesive Subgraphs* (HCS in short), because cohesive subgraphs are often with many important practical applications in network analysis [12]. It is easy to verify that the classic clique subgraph (completed subgraph) is a kind of HCS, as clique is cohesive and any subgraph of a clique is also a clique. However, since the strict constraint of clique model, many relaxed clique models are often used in practical applications. In this work, we focus mainly on two widely-used relaxed clique models, s-defective clique (s-dclique in short) [72] and s-plex [57], which also satisfy the hereditary property.

Definition 2.2 (s-dclique [72]). Given a graph G and a vertices set $Q \subseteq V$, $G(Q)$ is a s-dclique if $\overline{m}(Q) \leq s$.

Definition 2.3 (s-plex [57]). Given a graph G and a vertices set $Q \subseteq V$, $G(Q)$ is a s-plex if $\forall u \in Q, \overline{m}(u, Q) \leq s$.

It is worth mentioning that Definition 2.3 is slightly different from the traditional k-plex definition [57], as we exclude u when defining $\overline{m}(u, Q)$. In essence, the s-plex is equivalent to the $(k-1)$-plex based on the traditional definition [57].

By Definition 2.2 and Definition 2.3, it is easy to verify that both s-dclique and s-plex satisfy the hereditary property. Note that clique is a special case of s-dclique and s-plex (clique is a 0-dclique and 0-plex). Both s-dclique and s-plex are not naturally cohesive as clique [57]. However, as suggested in [57], we can easily make them cohesive by restricting their diameters no larger than 2. Fortunately, as shown in Lemma 2.4 and Lemma 2.5, very mild conditions can achieve this goal. Due to the space limits, all missing proofs can be found in the full version of this paper [39].

LEMMA 2.4. *A s-dclique with size q such that $q - 2 \geq s$ has diameter at most 2.*

LEMMA 2.5 ([57]). *A s-plex with size q that $q \geq 2s + 1$ has diameter at most 2.*

For practical applications, the value of s is often not very large (e.g., $s \geq 4$) [14, 21, 28, 72, 75]. When s is large, the s-dclique and s-plex are likely to loose their cohesiveness. Thus, the conditions in Lemma 2.4 and Lemma 2.5 are easy to meet. Note that although our frameworks and algorithms are designed for the s-dcliques and s-plexes that have diameter at most 2, they can be easily extended to the s-dcliques and s-plexes with arbitrary diameters.

For representation simplicity, we use (q, s)-dclique ((q, s)-plex) to represent an s-dclique (s-plex) with size q. Clearly, by our definition, both (q, s)-dclique and (q, s)-plex are HCSs. In this paper, we focus mainly on the problem of counting these two hereditary cohesive subgraphs in a graph.

Example 2.6. Fig. 1(a) illustrates an example graph G. G is a $(7, 2)$-plex and any subgraph of G is a 2-plex. In Fig. 1(b), the subgraph induced by $\{u_2, u_3, u_4, u_5, u_6\}$ is both a $(5, 1)$-dclique and $(5, 1)$-plex. The subgraph induced by $\{u_0, u_2, u_4, u_5, u_6\}$ (Fig. 1(c)) is a $(5, 1)$-plex. It is easy to derive that G contains 1 $(5, 1)$-dclique and 9 $(5, 1)$-plexes.

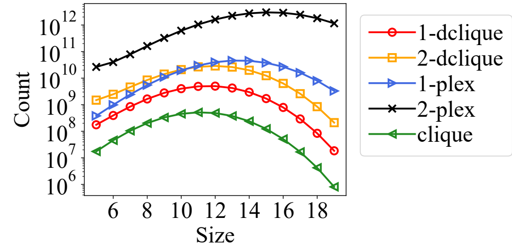


Figure 2: The counts of different kinds of HCS.

Algorithm 1: HCSList

Input: $G = (V, E)$, two integers q and s
Output: The count of hereditary cohesive subgraphs.
1 Let V' be ordered by degeneracy ordering $\{v_1, v_2, ...\}$;
2 **for** $i = 1$ *to* $|V'|$ **do**
3 $\quad C \leftarrow \vec{N}(v_i) \cup \vec{N}_2(v_i)$;
$\quad$ /* Construct the candidate set */
4 $\quad$ Listing$(C, \{v_i\})$;
5 **Procedure** Listing(C, R)
6 $\quad$ **if** $|R| = q - 1$ **then**
7 $\quad\quad$ $answer \leftarrow answer + |C|$;
8 $\quad\quad$ **return**;
9 $\quad$ **for** $u \in C$ **do**
10 $\quad\quad$ $C \leftarrow C \setminus \{u\}$;
11 $\quad\quad$ $C' \leftarrow \{v \in C | R \cup \{u\} \cup \{v\} is\ an$ HCS$\}$;
$\quad\quad$ /* Update the candidate set */
12 $\quad\quad$ Listing$(C', R \cup \{u\})$;

Problem Statement. Given a graph $G(V, E)$, parameters q, s and a kind of HCS ((q, s)-dclique or (q, s)-plex), our goal is to exactly count the number of HCSs in G.

Challenges. As discussed in [32, 33, 70], counting the number of q-cliques in a graph is an immensely challenging problem due to the exponential explosion in the number of q-cliques. Compared to the q-clique counting problem, the problem of counting the number of HCSs studied in this work is much more difficult, because the count of HCS is often orders of magnitude larger than the number of cliques. For example, the graph in Fig. 1(a) contains only two 4-cliques, while it contains twenty $(4, 1)$-dcliques and twenty-five $(4, 1)$-plexes. Fig. 2 plots the count of 1-dcliques, 1-plexes, 2-plexes and cliques on the Epinion network ($|V| = 75879$, $|E| = 811480$). The counts of s-dcliques and s-plexes are orders of magnitude larger than that of cliques. Moreover, existing approaches for counting q-cliques cannot be directly applied to HCS counting due to the increased complexity and additional constraints associated with HCSs.

3 THE LISTING-BASED SOLUTIONS

In this section, we first propose a general listing-based framework to count the HCSs. Then, based on this framework, we design specific algorithms for counting s-dcliques and s-plexes, along with some important carefully-designed pruning techniques.

3.1 A general listing framework HCSList

A simple approach to count HCSs in a graph is to list them. However, listing all HCSs in a graph can be challenging due to potential overlap and a potentially large number of HCSs. To address this problem, we propose a general backtracking algorithm, called HCSList, which can enumerates each HCS in a graph exactly once. The key idea of HCSList is described as follows. Since HCS satisfies the hereditary property, we can enumerate an HCS starting from a single vertex and maintaining a set R that represents the current sub-HCS. We then iteratively grow R to the desired size, and use a backtracking technique to list all HCSs.

Algorithm 1 gives the details of HCSList. The algorithm first orders the graph into a degeneracy ordering (line 1), which can be computed in $O(|E|)$ time using the core decomposition algorithm [6]. Based on the degeneracy ordering, the algorithm lists each HCS on the lowest-rank vertex of HCS using the Listing procedure (lines 2-4), which can guarantee that each HCS is only enumerated once. The Listing procedure maintains a candidate set C and a sub-HCS R, where each vertex in C can be added into R to form a larger sub-HCS. Initially, for each vertex v, Listing sets $R = \{v\}$ and constructs the candidates set C for v. Note that in this paper, we focus mainly on the HCS with a diameter less than 3, thus the initial candidate set is a set of all one-hop and two-hop outgoing neighbors of v (line 3). Then, for each v, the Listing procedure lists all HCSs that contains v as the lowest-rank vertex in the next backtracking call (lines 5-12). Line 11 utilizes the hereditary of HCS that $R \cup \{u\} \cup \{v\}$ are contained in an HCS which means that $R \cup \{u\} \cup \{v\}$ is an HCS itself. Finally, it is not necessary to enumerate C when $|R| = q-1$ because $R \cup \{u\}$ is an HCS for all $u \in C$ (lines 6-8). It is easy to show that HCSList only enumerates each HCS once, thus the correctness of the algorithm can be trivially guarantee.

Complexity analysis of HCSList. Theorem 3.1 gives the time and space complexity analysis of Algorithm 1. Note that for the time complexity, we mainly analyze the size of the recursion tree of HCSList, as its time complexity is mainly dominated by the recursion tree size.

THEOREM 3.1. *The time complexity of Algorithm 1 is $O(|V|\frac{\delta^2!}{(\delta^2-q)!}\tau)$, and the space complexity of Algorithm 1 is $O(|V|+|E|+q\delta^2)$, where δ is the degeneracy of the input graph and τ is the time consumed in each tree node of the recursion tree.*

Note that HCSList is a general framework. The parameter τ in Theorem 3.1 depends on specific HCSs, and we will give detailed analyses for two HCSs, including s-dclique and s-plex, in the following sections. Additionally, when counting s-dclique and s-plex specifically, we need to take into account of the details of design and implementation. The key is how to design effective prune strategies to reduce the search space. Below, we propose several effective and carefully-designed pruning techniques for both s-dclique and s-plex counting based on the HCSList framework.

3.2 Listing-based s-dclique counting

In order to transform the HCSList framework into an efficient algorithm for counting s-dcliques, we still need to develop effective pruning techniques that can enhance efficiency by leveraging the intrinsic properties of s-dcliques.

Pruning techniques for s-dclique counting. Our first pruning technique is designed to reduce the candidate set C before enumeration. Specifically, we aims to reduce the size of $\vec{N}(v_i) \cup \vec{N}_2(v_i)$. Below, we presents two useful lemmas which will be used to reduce $\vec{N}(v_i)$ and $\vec{N}_2(v_i)$ respectively.

LEMMA 3.2. *For each vertex $u \in \vec{N}(v_i)$ (v_i is in lines 2-4 of Algorithm 1), if u and v_i are contained in a (q,s)-dclique, then we have $|\vec{N}(u) \cap \vec{N}(v_i)| \geq q-s-2$.*

Lemma 3.2 shows that for an s-dclique with size q that contains v_i, the subgraph induced by all the vertices in $\vec{N}(v_i)$ must be a $(q-s-2)$-core. As a result, for each v_i in line 3 of Algorithm 1, we can reduce $\vec{N}(v_i)$ by computing the $(q-s-2)$-core on the subgraph induced by $\vec{N}(v_i)$.

LEMMA 3.3. *For each vertex $u \in \vec{N}_2(v_i)$, if u and v_i is contained in a s-dclique with size q, then u has at least $q-s-1$ neighbors in $\vec{N}(v_i)$.*

By Lemma 3.3, we can further prune the candidate set C by removing the vertices that have less than $q-s-1$ neighbors in the $(q-s-2)$-core of $\vec{N}(v_i)$ from $\vec{N}_2(v_i)$.

Our second pruning technique is to eliminate unnecessary branches of the procedure Listing in advance. However, how do we determine if a branch of Listing is unnecessary without actually accessing it? Below, we propose an upper bounding technique to solve this issue.

Assuming the procedure Listing is in the state between line 10 and line 11 of Algorithm 1, where u is already removed from C. We need to determine whether the following branch (line 12) can be pruned based on the values of u, R, and C. Let the vertices in $N(u) \cap C$ be ordered according to the count of non-neighbors in R, from smallest to largest, as $\{v_1, v_2, ...\}$. It follows that $\overline{m}(v_i, R) \leq \overline{m}(v_{i+1}, R)$. We can define $\omega(u,R,C) = \max_i\{\sum_{j\leq i}\overline{m}(v_j,R) \leq s - \overline{m}(R\cup\{u\})\}$. Since $\{v_1, v_2, ...\}$ is ordered, $\omega(u,R,C)$ is an upper bound on the number of vertices in $N(u) \cap C$ that can be added into $R \cup \{u\}$. With this information, we can calculate the total upper bound $\gamma_d(u,R,C)$ which can be used to prune the branch in advance if $\gamma_d(u,R,C) < q$. Lemma 3.4 provides further details on this process.

LEMMA 3.4. *Let $\gamma_d(u,R,C) = |R \cup \{u\}| + \min\{s - \overline{m}(R \cup \{u\}), \overline{m}(u,C)\} + \omega(u,R,C)$. Then, $\gamma_d(u,R,C)$ is an upper bound on the size of the s-dclique that the enumeration branch of Algorithm 1 can potentially reach.*

It is worth remarking that the work which focused on the maximum s-dclique problem [28] also defines an upper bound. However, their upper bound uses $|N(u) \cap C|$ directly instead of our $\omega(u,R,C)$. Thus, our upper bound can be deemed as a tighter bound which may also be more effective than the one defined in [28] for solving the maximum s-dclique problem.

Implementations. In the implementation of the s-dclique counting algorithm, we maintain an array A of size $|V|$ that $A[v]$ stores the value of $\overline{m}(v,R)$. When u is added into R, for each $v \in C' \setminus N(u)$, $A[v]$ should increase 1. Correspondingly, $A[v]$ should decrease 1 after the removal of u (line 12 of Algorithm 1). Clearly, the maintenance of A costs linear time. With the help of the array A, we can get the count of non-neighbors of each vertex in constant time.

The computation of the upper bound $\omega(u,R,C)$ does not really sort the set $N(u) \cap C$. Instead, we construct a bucket array B where $B[i]$ is the count of vertices in $N(u) \cap C$ that have i non-neighbors in R. Based on B, it is easy to derive that the time complexity of the computation of $\omega(u,R,C)$ is only $O(s)$.

Time complexity. In the worst case, the number of recursion tree nodes of the listing-based s-dclique counting algorithm is consistent with Theorem 3.1. Theorem 3.5 gives the time complexity of a specific recursion tree node with candidate set C.

THEOREM 3.5. *The time complexity in each recursion node of the listing-based s-dclique counting algorithm is $O(|C|^2)$.*

In addition, the time cost taken by the candidate set pruning techniques can be dominated by $O(\delta|E|)$. This is because in our pruning technique, we need to construct a subgraph induced by $\vec{N}(v_i)$ for each v_i, which takes at most $O(\delta|E|)$ in total using a technique developed in [24]. Since the total size of all the neighborhood subgraphs can be bounded by $O(\delta|E|)$, the total time complexity to compute the $(q-s-2)$-cores for all v_i is $O(\delta|E|)$.

3.3 Listing-based s-plex counting

Similar to s-dclique, designing an efficient s-plex counting algorithm based on Algorithm 1 also needs to design prune techniques.

Pruning techniques for s-plex counting. Like s-dclique, we can also use k-core based pruning technique to reduce the candidate size for s-plex counting. Specifically, we can first compute a $(q-s-2)$-core on the subgraph induced by $\vec{N}(v_i)$ without missing any s-plex, based on the results established in [19].

LEMMA 3.6 ([19]). *Assume that u and v are two vertices in a (q,s)-plex, then u and v have at least $q-2s-2$ common neighbors if (u,v) is an edge in the graph, and have at least $q-2s$ common neighbors otherwise.*

With Lemma 3.6, we can remove unpromising vertices from the candidate set C in line 3 of Algorithm 1. Specifically, we first reduce $\vec{N}(v_i)$ to a $(q-s-2)$-core by removing the vertices $\{u|u \in \vec{N}(v_i),\ |\vec{N}(u) \cap \vec{N}(v_i)| < q-s-2\}$. Then, we delete the vertices in $\vec{N}_2(v_i)$ that have degree less than $q-2s$ in the $(q-s-2)$-core. Finally, we combine the remaining vertices to get the updated candidate set without losing any (q,s)-plex.

To reduce the enumeration branches, we also devise an upper bound on the size of the s-plex that the current branch can access. Our upper bound is inspired by [21]. Specifically, when the Listing procedure is in the state between line 10 and line 11 of Algorithm 1, we calculate an upper bound which is defined as $\gamma_p(u,R,C) = |R \cup \{u\}| + \max_i\{\sum_{j \le i} \overline{m}(v_j,R) \le \sum_{v \in R}(s - \overline{m}(v,R))\} + \min(s - \overline{m}(u,R), \overline{m}(u,C))$, where $\{v_1, v_2, ...\}$ is the set $N(u) \cap C$ ordered according to $\overline{m}(v_i,R)$. This upper bound contains three parts: (1) $|R \cup \{u\}|$, the listed result, (2) $\max_i\{...\}$, the maximum count of neighbors of u, and (3) $\min(s - \overline{m}(u,R), \overline{m}(u,C))$, the maximum count of non-neighbors of u. It is easy to derive that $\gamma_p(u,R,C)$ is valid upper bound. If this upper bound is less than q, we can prune the current branch as it will not produce a valid s-plex.

Implementations. In the implementation of the listing-based s-plex counting algorithm, we maintain an array As with size $|V|$. For $v \in C$, $As[v]$ is exactly the set of vertices $R \setminus N(v)$. Since the count of the non-neighbors of v is at most s, As takes at most $O(s|V|)$ space. Similar to s-dclique, the maintenance of As takes $O(|C|)$ time for each $u \in C$. Specifically, u should be inserted into $As[v]$ for $v \in C \setminus N(u)$ if u is added into R and removed otherwise. With the array As, we can get the non-neighbors of each vertex efficiently.

Note that an efficient implementation of checking whether $R \cup \{u\} \cup \{v\}$ is a s-plex (line 11 of Algorithm 1) is nontrivial. A straightforward implementation is that for each $v \in C$ and for each $w \in R \cup \{u\} \setminus N(v)$, we verify whether $\overline{m}(w, R \cup \{u\} \cup \{v\}) \le s$. The drawback of this implementation is that it needs to check the non-neighbors of each vertex in C. To improve this, we observe that for each $w \in R \cup \{u\} \setminus N(v)$, only the vertices that $\overline{m}(w, R \cup \{u\}) = s$ do not meet the condition because w and v are non-adjacent. We can claim that w is also a non-neighbor of u. To see this, we assume to the contrary that $w \in N(u)$, then we have $\overline{m}(w,R) = s$. This is impossible because w is the non-neighbor of $v \in C$ and $\overline{m}(w, R \cup \{v\}) = s+1$ which is contradictory to the fact that $R \cup \{v\}$ is a s-plex. Therefore, our improved implementation is that for each $w \in R \setminus N(u)$, if $\overline{m}(w, R \cup \{u\}) = s$, we remove the non-neighbors of w in C. Note that in this improved implementation, it is no need to check the non-neighbors of the vertices in C.

Time complexity. For the s-plex counting algorithm, the number of the recursion-tree nodes is also consistent with Theorem 3.1. However, in each tree node, the time consumed for s-plex counting is higher than that for s-dclique counting. Theorem 3.7 shows the time complexity taken in each tree node for s-plex counting.

Algorithm 2: The new listing strategy

```
1  Procedure Listing(C, R)
2      if |R| = q − 1 then
3          answer ← answer + |C|;
4          return;
5      Split C into C1 and C2;
6      Listing(C1, R);
7      for u ∈ C2 do
8          C ← C \ {u};
9          C′ ← {v ∈ C|R ∪ {u} ∪ {v} is an HCS};
10         Listing(C′, R ∪ {u});
```

THEOREM 3.7. *The time complexity in each recursion-tree node of the listing-based s-plex counting algorithm is $O(s|C|^2)$.*

By Theorem 3.5 and 3.7, the listing-based s-plex counting algorithm is more sensitive to the parameter s, compared to the listing-based s-dclique counting algorithm. As confirmed in our experiments (see Table 2), with the increase of s, the running time of the s-plex counting algorithm increases more significantly compared to the s-dclique counting algorithm.

4 THE PIVOT-BASED SOLUTIONS

Although we develop many pruning techniques, the HCSList framework is still not very efficient for a relatively large q due to the exponential explosion of the number of HCS in real-world graphs. For example, the DBLP network has 2.8×10^8 $(5,1)$-dcliques, but it has 9.0×10^{13} $(10,1)$-dcliques. The sheer quantity of HCS makes listing-based algorithms infeasible. Another disadvantage of the HCSList framework is that it has redundant calculations. For example, two branches of the Listing procedure may have large overlapping sets of vertices, resulting in unnecessary duplicated calculations. A natural question that arises is whether it is possible to merge different branches of the Listing procedure to reduce redundant calculations. Additionally, is it possible to obtain the counts of HCS in a combinatorial way, rather than listing each individual one? We propose a novel pivot-based framework that addresses these questions.

To aid understanding, we will first introduce a new listing strategy for the pivot-based framework. Then, we explain the key pivoting technique for implementing combinatorial counting. Finally, based on the pivoting technique, we devise specific algorithms for counting s-dcliques and s-plexes with various optimization techniques.

4.1 Warm up: a new listing strategy

The new listing strategy utilize the idea of divide-and-conquer. Unlike the Listing procedure in Algorithm 1, it divides the candidate set into two parts: one part is used directly as the candidate set for the next level of recursion, while the other part is used for listing, similar to the approach used in HCSList. Algorithm 2 describes such a new listing strategy.

In Algorithm 2, the candidate set is split into two sets, C_1 and C_2 (line 5). C_1 is the candidate set of the next recursive call directly (line 6) and C_2 is to list each vertex (lines 7-10). The following theorem shows that such a new listing strategy can also correctly count all HCSs.

THEOREM 4.1. *Algorithm 2 does not miss any* HCS.

Note that the new listing strategy does not combine similar branches or eliminate redundant computation compared to the Listing procedure in HCSList. It still requires listing each HCS. Indeed, the

following theorem shows that the worst-case time and space complexity of Algorithm 2 is equal to those of the Listing procedure in Algorithm 1.

THEOREM 4.2. *The worst-case time and space complexity of Algorithm 2 is the same as those of the* Listing *procedure in Algorithm 1.*

Compared to the Listing procedure in Algorithm 1, a useful feature of Algorithm 2 is that it classifies all HCSs into three distinct categories (except the vertices contained in R):

(I) the HCS only containing vertices in C_1;
(II) the HCS only containing vertices in C_2;
(III) the HCS containing vertices both in C_1 and C_2.

With this classification, we are able to develop a pivot-based technique that can significantly reduce redundant calculations by computing the count of the class-III HCSs in a combinatorial manner.

Remark. It is worth remarking that the listing procedure of Algorithm 1 can be considered as a special case of Algorithm 2. When C_1 in Algorithm 2 is always empty during the recursion procedure, Algorithm 2 degenerates to the listing procedure of Algorithm 1.

4.2 A general pivot-based counting framework

As described previously, the HCS of class-III contains a sub-part in C_1. By virtue of being hereditary, this sub-part is also an HCS. The sub-HCS must be of class-I because it only contains vertices in C_1. Note that the recursive call in line 6 of Algorithm 2 has already searched all class-I HCSs, thus it is no need to repeatedly list such sub-HCSs when counting the class-III HCSs. Inspired by this, we propose a novel pivoting technique to improve the efficiency of counting the HCSs of class-III.

Definition 4.3 (pivot vertex). Let $\mathbb{H}$ be the set of HCSs in C_1 that $\forall H \in \mathbb{H}, R \cup H$ *is an* HCS. A vertex $u \in C_2$ is called a pivot vertex if $\forall H \in \mathbb{H}, R \cup H \cup \{u\}$ is an HCS.

Let u_p denote a pivot vertex. By utilizing $\mathbb{H}$, we can obtain the set of HCSs of class-III $\{H \cup \{u_p\} | H \in \mathbb{H}\}$ without actually listing them. It is straightforward to deduce that the number of HCS of size $q+1$ in $\{H \cup \{u_p\} | H \in \mathbb{H}\}$ is equivalent to the number of HCS of size q in $\mathbb{H}$. Thus, by combining these calculations, we can improve the computational efficiency.

Based on this idea, we propose a new counting framework, which is detailed in Algorithm 3. In addition to the candidate set C and the sub-HCS R, the Listing procedure in Algorithm 3 also maintains a vertex set D. The set D contains all the pivot vertices selected so far. Initially, $D = \emptyset$ and the candidate set C is divided into two sets C_1 and C_2 (line 8). If there exists a vertex u_p that can serve as a pivot vertex, it is removed from C_2 and added to D (lines 9-11). If there is no pivot vertex, the process proceeds in the same manner as Algorithm 2 (line 13). Once the size of R reaches $q-1$, each vertex in D and C can be added to R to generate a new HCS (lines 2-3). When C is empty, every $q - |R|$ vertices in D combined with R form a new HCS (lines 5-6).

By replacing the Listing procedure in HCSList with the Listing procedure outlined in Algorithm 3, we obtain a new general framework, called HCSPivot. Note that the pruning techniques proposed in Section 3 can be directly applied to HCSPivot. Below, we analyze the correctness of HCSPivot.

Correctness of the HCSPivot **framework.** Clearly, the backtracking process of Algorithm 3 can be represented as a recursion tree, where the root node has $D = \emptyset$ and the leaves have either $|R| = q-1$ or $C| = 0$. It is important to note that each HCS lies on exactly one

Algorithm 3: The pivot-based counting framework

```
1  Procedure Listing(C, R, D)
2      if |R| = q − 1 then
3          answer ← answer + |D| + |C|;
4          return;
5      if |C| = 0 then
6          answer ← answer + (|D| choose q−|R|);
7          return;
8      Split C into C1 and C2;
9      if there exists a pivot vertex u_p then
10         C2 ← C2 \ {u_p};
11         Listing(C1, R, D ∪ {u_p});
12     else
13         Listing(C1, R, D);
14     for u ∈ C2 do
15         C ← C \ {u};
16         C' ← {v ∈ C | R ∪ {u} ∪ {v} is an HCS};
17         Listing(C', R ∪ {u}, D);
```

path of the recursion tree. Let us label the recursive calls of Listing in line 11, line 13, and line 17 as L_1, L_2, and L_3, respectively. Each node in the recursion tree either has a child node from L_1 or L_2, and multiple child nodes from L_3. If an HCS consists only of vertices in C_1, it will be in the path down to either L_1 or L_2. If an HCS contains a pivot vertex u_p and all other vertices are in C_1, it will be in the path down to L_1. Finally, if an HCS contains vertices in C_2 (excluding u_p if it exists), it will be in the path down to L_3. These include all cases, and it is clear that each HCS can occur in only one of them. Thus, we can claim that HCSPivot is capable of accurately computing the count of HCSs. Theorem 4.4 formally proves that HCSPivot is correct.

THEOREM 4.4. HCSPivot *correctly counts the* HCSs.

Note that if no pivot vertex is selected in each recursion of Algorithm 3, Algorithm 3 degenerates to Algorithm 2. Thus, the time complexity of HCSPivot is the same as that of HCSList in the worst case. However, for the two specific HCSs considered in this paper, i.e., s-dclique and s-plex, we can always select valid pivot vertices in most recursions (for s-dclique counting, we can also guarantee that the pivot vertex always exists in each recursion), which can substantively boost the performance of HCSPivot. Indeed, as shown in our experiments, HCSPivot can be up to 7 orders of magnitude faster than HCSList based on such a powerful pivoting technique.

4.3 Pivot-based s-dclique counting

Note that HCSPivot is a general framework for counting HCSs. When using it to count s-dcliques, it is need to solve two issues: (1) how to split C into C_1 and C_2, and (2) how to choose a pivot vertex.

Lemma 4.5, derived from Definition 4.3, provides the criteria for a vertex u to be a pivot vertex for s-dclique counting.

LEMMA 4.5. *Let $\mathbb{H}$ be the set of s-dcliques in C_1 such that $\overline{m}(R \cup H) \leq s$ for each $H \in \mathbb{H}$, and u be a vertex in C_2. If $\overline{m}(R \cup H \cup \{u\}) \leq s$ for each $H \subseteq \mathbb{H}$, then u is a pivot vertex.*

A basic pivoting technique. By Lemma 4.5, it is easy to see that if $\overline{m}(u, R \cup H) = 0$ for all $H \in \mathbb{H}$, the vertex u must be a pivot vertex because $\overline{m}(R \cup H \cup \{u\}) = \overline{m}(R \cup H) + \overline{m}(u, R \cup H) \leq s$. Based on this observation, a basic pivoting method is to select a vertex u that $\overline{m}(u, R) = 0$ and $\overline{m}(u, H) = 0$. The pivot vertex u can be selected from the common neighbors of R and C_1 is set as $N(u) \cap C$. Here we take the maximum $|N(u) \cap C|$ over all u to ensure that the set C_1 is as large as possible. This is because a large C_1 will result in a small C_2 which can reduce the number recursive branches in the recursion

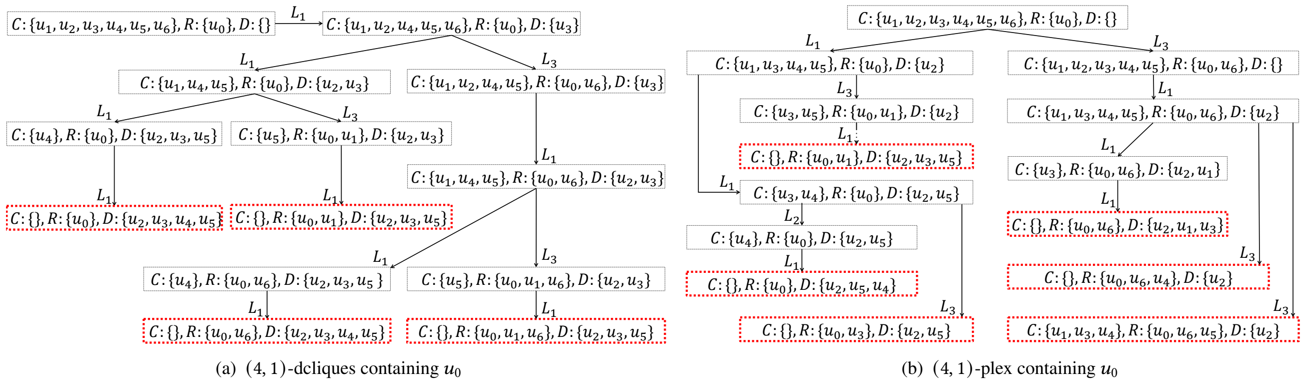


Figure 3: Illustration of the recursion tree of Algorithm 3 on the graph in Fig. 1(a).

tree of Algorithm 3, thus improving the efficiency of the algorithm. However, such a straightforward method has two limitations. First, it requires the pivot must be a common neighbor of R which is very restrictive and such a pivot may not exist (the common neighbor of R may be empty). Second, even though the pivot vertex has the maximum degree among the common neighbors of R, $|C_2|$ may still be large, meaning that there will be many recursive branches generated by the vertices in C_2, which may also slow down the algorithm.

An improved pivoting technique. To overcome the limitations of the basic pivoting method, we propose an improved pivoting technique which can ensure that the pivot vertex always exists in each recursion. Specifically, we relax the restriction $\forall H \subseteq \mathbb{H}, \overline{m}(R \cup H \cup \{u\}) \leq s$ in Lemma 4.5 to $\forall H \subseteq \mathbb{H}, \overline{m}(H \cup \{u\}) \leq s$. This change means that we do not consider the non-neighbors of u in R. Instead of choosing vertices in the common neighbors of R, we directly select the vertex with maximum degree in C as the pivot vertex and let $C_1 = N(u) \cap C$. This may result in the problem of $\exists u \in D, \overline{m}(u, R) > 0$ at the leaf of the recursion tree (lines 5-7 of Algorithm 3). Recall that for the basic pivoting technique, it has $\forall u \in D, \overline{m}(u, R) = 0$ since u is chosen from the common neighbors of R. Thus, we can choose arbitrary $q - |R|$ vertices from D. However, now it may occurs the case that $\overline{m}(R \cup H) > s$ where $|H| = q - |R|, H \subseteq D$. We need to compute how many subsets with size $q - |R|$ in D are valid answers. Fortunately, this is not a complicated issue, since $G(D)$ is always a clique as described in the following Theorem 4.6.

THEOREM 4.6. *If the vertex u is selected as the pivot vertex and let $C_1 = N(u) \cap C$, $G(D)$ must be a clique for the parameter D in each recursion node of Algorithm 3.*

Note that with our improved pivoting technique, the task of choosing arbitrary $q - |R|$ vertices from D when $|C| = \emptyset$ (as stated in line 6 of Algorithm3) transforms into computing the number of subsets H with $H \subseteq D$ and $|H| = q - |R|$ such that $H \cup R$ is a valid (q, s)-dclique.

THEOREM 4.7. *If $\sum_{u \in H} \overline{m}(u, R) \leq s - \overline{m}(R)$ holds, $H \cup R$ is a valid (q, s)-dclique for $H \subseteq D, |H| = q - |R|$.*

As a consequence, the problem of counting all valid (q, s)-dcliques is equivalent to a classic variant of *0-1 Knapsack Problem*, where the knapsack size is $s - \overline{m}(R)$ and the item weight is $\overline{m}(u, R)$ for each $u \in D$. This can be solved in $O(s|D|^2)$ time and $O(s|D|)$ space using *Dynamic Programming* [42].

The correctness of the pivot-based s-dclique counting algorithm can be guaranteed by Theorem 4.4 and Theorem 4.6. In each recursion, the time taken for selecting a pivot vertex is at most $O(|C|^2)$, the time consumed for computing the (q, s)-dclique counts when $|C = \emptyset|$ is $O(s|D|^2)$, and the time spent on lines 14-17 of Algorithm 3 is $O(|C^2|)$ (Theorem 3.5). Thus, the worst-case time complexity in each recursion is $O(\max(|C|^2, s|D|^2))$. Note that the practical performance of our pivot-based algorithm is extremely faster than the worst-case bound thanks to the benefits of the new pivot vertex selection strategy: (1) By selecting the vertex with the maximum degree in C as the pivot vertex, the size of $C_1 = N(u_p) \cap C$ is maximized and the number of vertices to list is minimized; (2) Pivot-vertices always exist at each node of the full recursion tree, and thus more s-dcliques are counted in a combinational way. The following example illustrates how our algorithm works.

Example 4.8. Fig. 3(a) depicts a sub-tree of the total recursion tree generated by Algorithm 3 on counting the $(4, 1)$-dcliques that include vertex u_0. The calls to Listing in line 11, line 13, and line 17 of Algorithm 3 are labeled as L_1, L_2, and L_3, respectively. The root node has a candidate set of $C = \{u_1, u_2, \ldots, u_6\}$, a sub-dclique of $R = \{u_0\}$, and an empty set $D = \{\}$. The vertex u_3 is selected as the pivot vertex because it has the maximum degree. The pivot vertex is then added to the set D and the candidate set C becomes $\{u_1, u_2, u_4, u_5, u_6\}$. The vertex u_2 has the maximum degree in the updated candidate set, so it is processed through L_1 and the vertex u_6 is processed through L_3. The last step is to solve a *0-1 Knapsack Problem* at the leaf node. Let us consider the example of the leaf node $R = \{u_0\}$, $D = \{u_2, u_3, u_4, u_5\}$. The problem is to choose $q - |R| = 3$ vertices from D. Since $G(D)$ must be a clique, we need to ensure that the three chosen vertices have at most $s - \overline{m}(R) = 1$ non-neighbors in R. Clearly, the two correct answers are $\{u_2, u_3, u_5\}$ and $\{u_2, u_4, u_5\}$.

4.4 Pivot-based s-plex counting

Compared to counting s-dcliques, counting s-plexes is a more complicated task as each vertex is allowed to miss at most s edges. Lemma 4.9, which derives from Definition 4.3, provides the criteria for a vertex u to be considered as a pivot vertex.

LEMMA 4.9. *Let $\mathbb{H}$ be the set of s-plex in C_1 that $\forall H \in \mathbb{H}, \forall v \in R \cup H, \overline{m}(v, R \cup H) \leq s$. Let u be a vertex that $u \in C_2$ and H' represents $R \cup H \cup \{u\}$ in short. If $\forall H \subseteq \mathbb{H}, \forall v \in H', \overline{m}(v, H') \leq s$, u is a pivot vertex.*

The condition $\forall H \subseteq \mathbb{H}, \forall v \in H', \overline{m}(v, H') \leq s$ means that for any $H \in \mathbb{H}$, adding the vertex u into the set $R \cup H$ always results in an s-plex. It encompasses both (1) the requirement that $\overline{m}(u, H') \leq s$, meaning that u itself has at most s non-neighbors, and (2) the requirement that $\forall v \in (R \cup H) \setminus N(u), \overline{m}(v, H') \leq s$, meaning that the non-neighbors of u still have at most s non-neighbors after the addition of u. Based on these observations, we propose a pivot-selection technique for counting s-plexes based on Theorem 4.10.

THEOREM 4.10. *Let $u \in C$, and let $C_1 = N(u) \cap C$. The vertex u can serve as a pivot vertex if $\forall v \in R \setminus N(u), \overline{m}(v, R \cup C_1) \le s-1$.*

According to Theorem 4.10, the steps of choosing the pivot vertex is as follows. First, for each $v \in C$, we evaluate whether all vertices in $R \setminus N(u)$ have at most $s-1$ non-neighbors in $R \cup (C \cap N(v))$. Then, we obtain a set of vertices that can serve as the pivot vertex. Among them, we choose the one that has maximum degree in C as the pivot vertex. If no vertex in C can serve as the pivot vertex, we set $C_1 = C \cap N(v)$ that v has maximum degree in C.

However, for each $v \in C$ and for each $w \in R \setminus N(v)$, computing the count of non-neighbors of w in $R \cup (C \cap N(v))$ is quite a heavy work. To enhance the efficiency, we improve the strategy of choosing the pivot vertex according to Corollary 4.11, which comes from Theorem 4.10.

COROLLARY 4.11. *Let u be a vertex in C and let $C_1 = N(u) \cap C$. u can serve as a pivot vertex if $\forall v \in R \setminus N(u), \overline{m}(v, R \cup C) \le s-1$.*

The correctness of Corollary 4.11 can be ensured by Theorem 4.10 because C_1 is a subset of C. According to Corollary 4.11, we just need to compute the count of non-neighbors in $R \cup C$ for each vertex in R, which can be done in linear time. Similarly, among the set of vertices that can serve as the pivot vertex, we choose the one that has maximum degree in C as the pivot vertex, and let C_1 be the set of neighbors of the pivot vertex. It is important to note that if the condition "$\forall v \in R \setminus N(u), \overline{m}(v, R \cup C) \le s-1$" does not hold, there is no pivot vertex can be selected. In this case, we just pick the maximum-degree vertex u in C and set $C_1 = N(u) \cap C$.

Based on this improved pivoting technique, the time taken for selecting a pivot vertex is $O(|C|^2)$ in each recursion. Clearly, computing the set C_1 consumes at most $O(|C|^2)$ time, and the time spent on lines 14-17 of Algorithm 3 is $O(s|C|^2)$ (Theorem 3.7). Thus, the total time cost in each recursion is $O(s|C|^2)$. Similar to the pivot-based s-dclique counting algorithm, the practical performance of the pivot-based s-plex counting algorithm is also much better than the worst-case time bound, as confirmed in our experiments.

Example 4.12. Fig. 3(b) is the recursion tree of Algorithm 3 on counting $(4,1)$-plex. We label the calls of Listing in line 11, line 13, line 17 of Algorithm 3 as L_1, L_2, L_3, respectively. The root node has $C = \{u_1, u_2, ..., u_6\}$ and $R = \{u_0\}$. Among the root node, u_2, u_5 and u_6 can serve as the pivot vertex and they all have 4 neighbors in C. The vertices u_1, u_3 and u_4 cannot serve as the pivot vertex. This is because they have non-neighbors in $C \cup R$ and the count of the non-neighbor in $C \cup R$ is at most $s-1=0$ according to Corollary 4.11. Suppose that choose u_2 as the pivot vertex, and then $C_1 = N(u_2) = \{u_1, u_3, u_4, u_5\}, C_2 = \{u_6\}$. C_1 is the candidate set of the child node through L_1. The vertex u_6 is inserted into R in the child node through L_3. Note that on the recursion node with $C = \{u_3, u_4\}$, $R = \{u_0\}$, and $D = \{u_2, u_5\}$, no vertex can serve as the pivot vertex. In this case, we set $C_1 = C \cap N(u_3) = \{u_4\}$ and $C_2 = \{u_3\}$. Here a child node through L_2 with candidate set C_1 occurs, and u_3 is inserted into R in the child node through L_3.

In Fig. 3(b), the red dotted leaves are the answers. The recursion node with $C = \{u_1, u_3, u_4\}$, $R = \{u_0, u_6, u_5\}$, and $D = \{u_2\}$ is the leaf node because $|R| = q-1$ (line 2 of Algorithm 3). In the red dotted leaves, the combination of any $q-|R|$ vertices in D with R forms a $(4,1)$-plex. For instance, in the leaf node with $R = \{u_0, u_6\}$ and $D = \{u_2, u_1, u_3\}$, every two vertices of D combined with R will result in a $(4,1)$-plex.

4.5 Discussions

Counting HCS with size in a range $[q_l, q_r]$ simultaneously. A striking feature of Algorithm 3 is that it is highly adaptable and capable of simultaneously computing the counts of HCSs of different sizes in a range of $[q_l, q_r]$. This is achievable due to the fact that the recursion tree of Algorithm 3 is only slightly impacted by the parameter q. By making a few modifications to the algorithm, we can count HCSs with sizes in the range $[q_l, q_r]$ simultaneously. These modifications include: (1) modifying $|R| = q-1$ in line 2 to $|R| = q_r - 1$, and (2) modifying $answer \leftarrow answer + \binom{|D|}{q-|R|}$ in line 6 to $answer_q \leftarrow answer_q + \binom{|D|}{q-|R|}$ for every $q \in [q_l, q_r]$. Note that the listing-based framework, i.e., Algorithm 1, cannot simultaneously compute the counts of HCSs of different sizes in a range. In the experiments, we will use the counts of HCSs with various size to build a *graph profile* [29, 68] for a given network. The experimental results show the ability of the proposed *graph profile* to characterize different types of networks.

Local counting. Another important feature of Algorithm 3 is that it can obtain the local count of HCS for each vertex or edge. Here local count refers to the number of HCSs that contain a specific vertex or edge. For a given vertex u or edge (u,v), we define its local count as c_u or $c_{(u,v)}$, respectively. To implement locally counting using Algorithm 3, we only need to add a counter for each vertex (or each edge) in line 6. More specifically, to compute the local counts for vertices, we partition the vertices into two categories: $u \in R$ and $u \in D$. If $u \in R$, then c_u is updated as $c_u \leftarrow c_u + \binom{|D|}{q-|R|}$. On the other hand, if $u \in D$, then c_u is updated as $c_u \leftarrow c_u + \binom{|D|-1}{q-|R|-1}$. To calculate the local counts for edges, we divide the edges into three categories: (1) ($u \in R, v \in R$), (2) ($u \in R, v \in D$), and (3) ($u \in D, v \in D$). For category (1), $c_{(u,v)}$ is updated as $c_{(u,v)} \leftarrow c_{(u,v)} + \binom{|D|}{q-|R|}$. For category (2), $c_{(u,v)}$ is updated as $c_{(u,v)} \leftarrow c_{(u,v)} + \binom{|D|-1}{q-|R|-1}$. Finally, for category (3), $c_{(u,v)}$ is updated as $c_{(u,v)} \leftarrow c_{(u,v)} + \binom{|D|-2}{q-|R|-2}$. Note that the listing-based framework, i.e., Algorithm 1, cannot compute the local counts for all vertices (or edges) in such a combinatorial manner. In the experiments, we will apply the local counts to construct a matrix M, where M_{ij} is the count of HCS containing the edge (i,j). Such a matrix M is then used for graph clustering applications which achieves much better performance compared to the state-of-the-art graph clustering methods, as shown in our experiments.

Difference with the maximal s-dclique or s-plex enumeration. Note that existing maximal s-dclique or s-plex enumeration algorithms [20, 21, 28, 66, 75] also use a pivoting technique to reduce the enumeration branches. Our pivot-based algorithm (i.e., Algorithm 3), however, is significantly different from those existing algorithms. First, the recursion tree of Algorithm 3 is completely different from those of the maximal s-dclique or s-plex enumeration algorithms. Specifically, the leaf nodes of our recursion tree may not be maximal results. However, for the maximal s-plex or s-dclique enumeration algorithms [20, 21, 28, 66, 75], the leaf nodes of the recursion tree are maximal s-plexes or s-dcliques. Second, to ensure that each HCS is counted exactly once, our algorithm guarantees that there are no overlapping sub-HCS among the HCSs at the leaf nodes. This differs from the maximal s-dcliques or s-plexes enumeration algorithms, which may contain overlapping sub-structures at the leaf nodes. Third, our definition of pivot vertex (Definition 4.3) is specifically designed for the HCS counting problem, which distinguishes it from

Table 1: Datasets

Networks	$\|V\|$	$\|E\|$	δ	Type
WikiV	7115	201524	53	Social network
Caida	26475	106762	22	Autonomous system network
Epinion	75879	811480	67	Social network
EmailEu	265009	728960	37	Communication network
Amazon	403394	4886816	10	Communication network
DBLP	425957	2099732	113	Co-authorship network
Pokec	1632803	44603928	47	Social network
Skitter	1696415	22190596	111	Autonomous system network

the maximal s-dcliques or s-plexes enumeration problems. Moreover, to improve the efficiency, we have devised unique and novel pivoting techniques to prune unnecessary candidate sets and branches (Theorem 4.6 and 4.10), which significantly differ from the pivoting techniques utilized in the enumeration of maximal s-dcliques or s-plexes.

Relation to the pivot-based q-clique counting algorithm. PIVOTER is the state-of-the-art pivot-based q-clique counting algorithm [33] which builds upon the classic Bron-Kerbosch algorithm for maximal clique enumeration [9]. Since q-clique is a type of HCS, our pivot-based algorithm (i.e., Algorithm 3) can also be applied to count q-cliques. We note that PIVOTER is a special case of Algorithm 3. Specifically, we remove lines 12-13, select the maximum-degree vertex in C as the pivot vertex v_p, and set $C_1 = N(v_p) \cap C$, then Algorithm 3 is exactly equivalent to the PIVOTER algorithm. Compared to PIVOTER, which can only be applied to count q-cliques, Algorithm 3 is a general framework which is capable of processing all HCS counting problems. Moreover, the idea of splitting the candidate set into C_1 and C_2 in Algorithm 3 is not used in PIVOTER [33], which is critical to develop a general approach to all HCS counting problems. Based on Algorithm 3, we need to design different candidate set partition and pivot vertex selection strategies when handling various HCS counting problems. Additionally, our algorithm incorporates several carefully-designed pruning techniques that are tailored to the specific restrictions defined in Definitions 2.2 and 2.3, which are also not shown in PIVOTER [33].

Relation between listing and pivot based solutions. Note that our listing-based solutions are both essential and nontrivial, and they are not always inferior to pivot-based solutions. In cases where q is a small constant, the time complexity of listing-based solutions is polynomial, while pivot-based solutions always have exponential time complexity for any value of q. Our experiments have also shown that listing-based solutions outperform pivot-based solutions in certain scenarios (refer to Table 2). In Section 4.1, the new listing strategy serves as a precursor to the subsequent pivot-based solutions. Its main purpose is to improve the understandability of the pivot-based solutions.

Extension to counting HCSs with a larger diameter. To apply Algorithm 1 and Algorithm 3 to count HCSs with diameter larger than 2, we need to modify line 3 of Algorithm 1 to include vertices in longer hops. As for Algorithm 3, we can make use of the basic pivot technique in Definition 4.3.

5 EXPERIMENTS

In this section, we conduct extensive experiments to evaluate the performance of the proposed solutions. In addition, we also evaluate the effectiveness of our algorithms by presenting case studies and demonstrating their applications in Section 5.3 and 5.4.

5.1 Experimental setup

Algorithms. We evaluate 4 algorithms, namely Dlist, Plist, Dpivot, and Ppivot. Dlist and Plist are listing-based approaches to count s-dclique and s-plex respectively. On the other hand, Dpivot and Ppivot are pivot-based algorithms used to count s-dclique and s-plex respectively. All algorithms are implemented in C++ and all of them are integrated with the pruning techniques developed in Section 3 as their default setting. Since this is the first work to study the problem of HCS counting, we use the list-based algorithms Dlist and Plist as the baselines.

Datasets. We selected 8 real-world networks to evaluate the performance of different algorithms. The details of the datasets can be found in Table 1. All the datasets used in our experiments are obtained from the SNAP project [36].

All the experiments are conducted on a server with an AMD 3990X CPU, 256GB memory, and Linux CentOS 7 operating system. Similar to previous studies on enumerating maximal s-dcliques [20] and s-plexes [21, 66, 75], we mainly evaluate different HCS counting algorithms with $s \leq 3$. This is because real-world applications often require that the subgraph pattern is cohesive and also very similar to clique [21, 66, 75]. When $s > 3$, the HCS may become sparse, making it more reasonable to consider small values of s, as previous studies have commonly done [20, 21, 66, 75].

5.2 Performance studies

Comparing the listing and pivot-based algorithms. Comparing the performance of Dlist and Dpivot in Table 2, we observe that they exhibit similar performance when q is small. For example, on WikiV, Dlist takes 10.2 seconds to count $(5, 1)$-dclique, while Dpivot needs 9.62 seconds. However, as q becomes larger, Dpivot can be several orders of magnitude faster than Dlist on many networks. For example, on DBLP, when $s = 1$ and $q = 9$, Dlist consumes 162301.4 seconds, while Dpivot takes only 0.1 seconds. We also find that on some networks, such as Caida and Amazon, both Dlist and Dpivot perform similarly even for a large q. This is because the count of HCS with size q in each network differs. If the count is large, Dlist is often much slower than Dpivot, while if the count is not very large, Dlist performs comparably with Dpivot. For instance, Epinion has 1.7×10^9 $(15, 1)$-dcliques, whereas Caida only has 3.2×10^4 $(12, 1)$-dcliques. Similarly, Ppivot is more efficient than Plist on complex networks when q is large. These results demonstrate the high efficiency of the proposed pivot-based counting algorithms.

Difference on counting s-dclique and s-plex. From Table 2, we observe that Plist is slower than Dlist, and Ppivot is slower than Dpivot for a given s and q. For instance, on Skitter when $s = 1$ and $q = 30$, Dpivot is an order of magnitude faster than Ppivot. This result suggests that s-plex is a more complex structure than the s-dclique, which is often more difficult to count.

Dpivot and Ppivot with various q and s. In Table 2, we observe that the running time of Dpivot and Ppivot decreases as q increases. However, the counter-intuitive thing is that the count of HCS may not decrease as q becomes larger. For instance, Skitter has 9.3×10^{13} $(10, 1)$-dcliques and 1.1×10^{21} $(30, 1)$-dcliques, but Dpivot runs faster on counting $(30, 1)$-dcliques than $(10, 1)$-dcliques. This phenomenon occurs because larger values of q can prune more candidate sets, thanks to the core-based and upper-bound based pruning techniques developed in Section 3. Note that the size of the search tree in Algorithm 3 is almost not affected by the parameter q. Thus, no matter what the value of q is, Algorithm 3 will enumerate all the

Table 2: Running time (sec) of different algorithms with various q and s

Networks	Running time (sec)																	
	$s = 1$						$s = 2$						$s = 3$					
	s-dclique			s-plex			s-dclique			s-plex			s-dclique			s-plex		
	q	Dlist	Dpivot	q	Plist	Ppivot	q	Dlist	Dpivot	q	Plist	Ppivot	q	Dlist	Dpivot	q	Plist	Ppivot
WikiV	5	10.2	**9.6**	5	18.3	**16.7**	7	**126.4**	134.9	7	4315.5	**3139.3**	10	3083.8	**720.3**	15	-	**70049.5**
	10	13.1	**5.7**	10	80.9	**14.6**	12	60.1	**43.5**	12	5059.8	**1618.3**	15	**95.3**	115.4	20	3612.8	**1548.9**
	15	**1.4**	1.8	15	8.1	**2.4**	17	**3.7**	7.8	17	-	**88.9**	20	**3.9**	9.4	25	**1.9**	2.1
Caida	6	0.2	**0.1**	6	0.3	**0.2**	7	0.9	**0.6**	7	19.3	**9.8**	7	22.9	**6.6**	7	-	**14131.6**
	9	0.1	**0.0**	9	0.3	**0.1**	9	0.6	**0.3**	9	12.9	**5.4**	9	3.6	**2.0**	9	-	**375.8**
	12	**0.0**	**0.0**	12	0.1	**0.0**	12	**0.1**	**0.1**	12	4.3	**1.1**	12	0.7	**0.4**	12	193.0	**90.7**
Epinion	5	48.2	**36.0**	5	**62.6**	66.3	7	1450.8	**588.9**	7	-	**14507.9**	10	-	**3747.7**	25	-	**93791.8**
	10	320.7	**25.8**	10	1813.7	**109.6**	12	-	**267.9**	12	-	**26011.0**	15	-	**1137.0**	30	-	**733.7**
	15	462.2	**12.9**	15	7951.5	**56.9**	17	6013.3	**92.6**	17	-	**12645.6**	20	1046.6	**240.7**	35	1.0	**0.9**
EmailEu	5	3.4	**1.9**	5	4.0	**3.7**	7	16.8	**14.8**	7	386.5	**281.4**	10	225.7	**48.1**	10	-	**14966.8**
	10	1.9	**0.7**	10	11.4	**1.7**	9	36.7	**4.4**	12	508.0	**113.6**	15	244.7	**6.6**	15	-	**3577.9**
	15	**0.2**	**0.2**	15	0.8	**0.3**	12	6.2	**3.5**	17	27.1	**7.7**	20	118.8	**0.4**	20	-	**77.5**
Amazon	6	2.8	**1.0**	6	4.2	**2.7**	7	6.0	**1.7**	7	39.5	**18.3**	7	53.1	**10.5**	7	-	**23008.3**
	9	0.7	**0.3**	9	0.9	**0.5**	9	1.5	**0.5**	9	7.1	**2.5**	9	5.3	**1.1**	9	254.7	**106.7**
	12	0.3	**0.1**	12	**0.1**	**0.1**	12	0.3	**0.1**	12	**0.2**	**0.2**	12	0.4	**0.2**	12	2.6	**1.0**
DBLP	5	4.9	**0.5**	5	6.1	**0.9**	7	1100.2	**0.7**	7	1518.9	**11.8**	7	1439.1	**5.0**	10	-	**930.6**
	7	1005.5	**0.2**	7	1192.8	**0.3**	12	-	**0.2**	12	-	**2.8**	9	182028.8	**1.8**	15	-	**550.6**
	9	162301.4	**0.1**	9	193757.0	**0.2**	17	-	**0.2**	17	-	**2.3**	12	-	**1.3**	20	-	**459.3**
Pokec	5	279.6	**139.7**	5	**493.8**	538.6	7	797.2	**493.1**	7	66895.4	**8683.0**	10	-	**936.9**	15	-	**54975.3**
	10	125.3	**22.4**	10	468.2	**57.1**	12	1209.2	**67.6**	12	16931.4	**1584.5**	20	2603.2	**14.5**	20	-	**4817.3**
	15	163.8	**10.0**	15	673.6	**13.7**	17	1075.8	**14.8**	17	14930.8	**238.3**	30	30.5	**4.8**	25	-	**119.8**
Skitter	10	-	**4209.7**	30	-	**54752.0**	30	-	**25884.6**	55	-	**49106.6**	40	-	**67878.3**	60	-	**423542.2**
	30	-	**1849.2**	40	-	**13224.9**	40	-	**7784.1**	60	-	**1069.3**	50	-	**7355.9**	65	-	**2177.8**
	50	-	**906.7**	50	-	**728.8**	50	-	**1201.2**	65	-	**96.3**	60	-	**668.9**	70	-	**136.2**

Table 3: The reduction rate of candidate pruning technique.

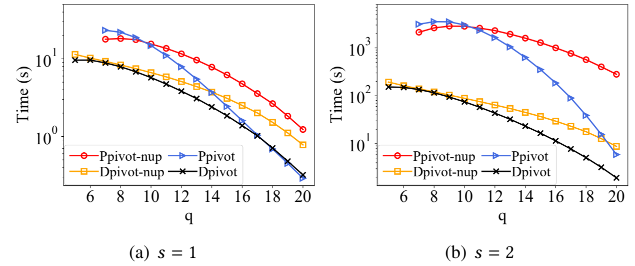
q	Networks	The reduced ratio of the candidate set					
		$s = 1$		$s = 2$		$s = 3$	
		s-dclique	s-plex	s-dclique	s-plex	s-dclique	s-plex
10	WikiV	95.33%	95.39%	94.72%	94.02%	93.95%	91.52%
	Epinion	95.81%	95.96%	95.30%	94.92%	94.64%	92.81%
	Amazon	94.83%	93.94%	94.07%	93.41%	93.28%	92.06%
	Pokec	97.66%	97.72%	97.36%	97.28%	97.01%	95.89%
20	WikiV	97.59%	97.67%	97.52%	97.44%	97.39%	97.19%
	Epinion	97.62%	97.66%	97.55%	97.51%	97.45%	97.28%
	Amazon	100%	100%	100%	100%	100%	100%
	Pokec	99.80%	99.72%	99.71%	99.47%	99.60%	99.06%

large HCSs. Therefore, the larger the value of q, the smaller the size of the candidate set, and the faster Dpivot and Ppivot are.

In Table 2, we can also observe that the running time of Dpivot and Ppivot increases as the value of s increases. For instance, on the WikiV network, Dpivot takes 9.62 seconds for $(5, 1)$-dclique and 134.9 seconds for $(7, 2)$-dclique, which is an increase of 14×. On the other hand, Ppivot takes 16.7 seconds for $(5, 1)$-plex and 3139.3 seconds for $(7, 2)$-plex, which is an increase of 188×. This is because the number of HCSs in a network increases significantly when the parameter s increases.

Evaluating the pruning technique: candidate reduction. Table 3 reports the reduction rate of the candidate pruning technique. q is fixed to 10 and 20. The reduction rate is computed as $\frac{c_{pre}-c_{now}}{c_{pre}}$, where c_{pre} is the total size of all the candidate set before pruning, i.e. $\sum_{v_i} |\vec{N}(v_i) \cup \vec{N}_2(v_i)|$ in line 3 of Algorithm 1 and c_{now} is the total size after pruning. The table shows that the candidate reduction technique can remove over 90% of unnecessary vertices, which is a significant speed-up. When q goes larger, more vertices can be removed from the candidate set. For example, when $s = 1$ and $q = 10$, the total candidate set size of WikiV after pruning on s-dclique counting is 3.5×10^5. When $s = 1$ and $q = 20$, the size becomes 2.0×10^4. Therefore, the candidate reduction based pruning technique is effective.

Evaluating the prune technique: upper bound. In Fig. 4, we compare Ppivot against Ppivot-nup, and compare Dpivot against Dpivot-nup on WikiV, where Ppivot-nup and Dpivot-nup are Algorithm 3 without the prune techniques based on the upper bounds.

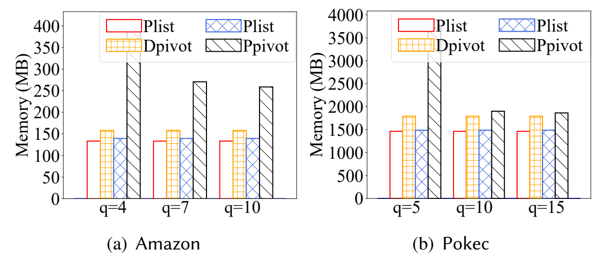


(a) $s = 1$ (b) $s = 2$

Figure 4: Effectiveness of the upper bounds (WikiV).

Table 4: The percentage of HCSs counted in combination

q	Networks	s-dclique		s-plex	
		$s = 1$	$s = 2$	$s = 1$	$s = 2$
7	WikiV	95.7%	91.8%	87.6%	57.4%
	Epinion	92.8%	87.3%	84.3%	51.8%
	Amazon	100.0%	100.0%	100.0%	98.4%
	Pokec	98.5%	96.5%	94.8%	72.1%
14	WikiV	100.0%	100.0%	100.0%	99.8%
	Epinion	100.0%	100.0%	100.0%	99.6%
	Amazon	100.0%	100.0%	100.0%	100.0%
	Pokec	100.0%	100.0%	100.0%	100.0%



(a) Amazon (b) Pokec

Figure 5: Memory overheads of different algorithms

Similar results can also be observed on the other datasets. As shown in Fig. 4, the effectiveness of the upper bounds increases when q becomes larger. The effect of s on the effectiveness of the upper bounds also increases with increasing s. For instance, by fixing $q = 20$, Ppivot-nup is 4× slower than Ppivot when $s = 1$ while 47× when $s = 2$.

Table 5: The running time of counting HCS with size in $[q_l, q_r]$ simultaneously ($q_l = 5, q_r = 20$).

Networks	Running time (sec) Dpivot q_l	Dpivot $[q_l, q_r]$	Ppivot q_l	Ppivot $[q_l, q_r]$
WikiV	9.62	11.11	16.68	32.27
Epinion	35.98	47.595	66.31	184.17
Amazon	1.02	1.86	4.72	4.74
Pokec	139.73	146.88	538.55	570.89

The effectiveness of the pivoting technique. In Algorithm 3, the HCSs are counted by listing (line 3) or counted in a combinatoric manner (line 6). Clearly, the more HCSs are counted in a combinatoric manner, the better the acceleration of the pivot-vertex technique. Table 4 shows the percentage of HCSs counted in a combinatoric manner. In Table 4, Dpivot has larger percentages than Ppivot. Moreover, Ppivot is more sensitive to the parameter s. This is because Dpivot always has a pivot vertex at each node of the recursion tree, while Ppivot depends on R, C and s as stated by Corollary 4.11. Table 4 also shows that the HCSs with larger size are more easy to count in a combinatoric way. With larger value of q, the condition $|R| = q - 1$ (line 2 in Algorithm 3) is harder to meet. In general, our pivot-based algorithms enable a large number of HCSs to be counted in a combinatoric manner. These results further confirm the high efficiency of the proposed pivot-based solutions.

Memory overheads. Fig. 5 compares the memory costs of the proposed algorithms on Amazon and Pokec given that $s = 1$. For other parameter settings and other datasets, the results are consistent. The pivot-based approaches require more memory compared to the listing-based solutions. This is because the recursive depth of the pivot-based algorithms is often deeper than the listing-based solutions, thus resulting in more space usages. However, we can also note that the memory overheads of the pivot-based solutions are still within the same order of magnitude as listing-based algorithms, due to the fact that both the listing-based and pivot-based methods are depth-first search algorithms which are typically space-efficient. For example, when $q = 4$, Plist consumes 140 MB on Amazon and Ppivot uses 394 MB on the same datasets.

Counting HCS with size in $[q_l, q_r]$ simultaneously. As described in Section 4.5, Dpivot and Ppivot can count HCS with size in a range simultaneously. Table 5 shows the running time, where the column q_l means the running time of counting only HCS with size q_l. The parameters are set as $s = 1$, $q_l = 5$ and $q_r = 20$. As shown in Table 5, The running time of counting $[q_l, q_r]$ and counting only q_l are within the same order of magnitude. The results with other parameters are similar. This is because counting $[q_l, q_r]$ and counting only q_l has the same size of the search tree and only has difference in the leaves nodes (as described in Algorithm 3 and Section 4.5). These results further demonstrate the advantages of the pivot-based algorithms, compared to the list-based algorithms.

Local counting. We compared the running time of global and local counting in Table 6, with parameters $s = 1$. Local counting includes counting HCSs in each vertex and edge. In Table 6, we find that the local-vertex counting is slightly slower than the globally counting. This is because the computation of local-vertex counting only requires scanning the vertices, as described in Section 4.5. However, the local-edge counting needs to scan the edges, and since the number of edges is larger than the number of vertices, the running time of local-edge counting is slower but still within the same order of magnitude. For example, in Table 6 with $q = 7$, local-edge counting is at most 2.42× slower than global counting. These results indicate

Table 6: The running time of locally counting

q	Networks	Running time (sec) Dpivot global	Dpivot vertex	Dpivot edge	Ppivot global	Ppivot vertex	Ppivot edge
7	WikiV	11.3	11.9	21.7	29.6	30.1	58.4
	Epinion	43.3	45.5	94.5	119.6	121.8	288.9
	Amazon	0.8	0.8	1.2	1.8	1.8	2.2
	Pokec	60.6	65.2	109.1	188.5	190.3	268.4
14	WikiV	3.0	4.7	7.0	4.7	4.6	6.8
	Epinion	19.2	28.1	72.5	86.4	86.5	194.5
	Amazon	0.1	0.1	0.1	0.1	0.1	0.1
	Pokec	11.6	14.1	22.5	18.6	18.6	27.2

that Dpivot and Ppivot are very efficient in locally counting HCSs for each vertex or edge.

5.3 Application 1 : HCS-based graph clustering

In this section, we show the application of the HCS counts for motif-based graph clustering. One of the most popular metrics in community detection is *conductance* [7, 27, 37, 54, 60, 69]. The conductance is the ratio of the count of edges leaving the community and the count of edges in the community. Similarly, the motif-based conductance [7, 60] is the ratio of the count of motifs leaving the community and the count of motifs in the community, which is formulated as $\Phi(S) = \frac{C_M(S:\overline{S})}{min(C_M(S), C_M(\overline{S}))}$, where $\overline{S}$ is the remainder vertices of S, $C_M(S)$ is the count of the motif M in $G(S)$ and $C_M(S:\overline{S})$ is the count of the motif M that contains both vertices in S and $\overline{S}$. $\Phi(S)$ measures how well a community S preserves the occurrences of M compared to its complement $\overline{S}$. A community with low motif-based conductance means that it preserves the occurrences of the motif M well within the community.

To minimize the motif-based conductance $\Phi(S)$, we can use the classic spectral clustering algorithm, as suggested in previous studies [7, 60]. Essentially, $\Phi(S)$ can be viewed as a normalized cut, where $C_M(S : \overline{S})$ represents the count of motifs in the graph cut $(S : \overline{S})$. To apply spectral clustering, we define a weight matrix $W_M \in N^{|V|\times|V|}$, where $(W_M)_{ij}$ is the count of motifs containing edge (i, j). Using this weight matrix, we can follow the same steps as the classic spectral clustering algorithm to obtain clusters that minimize the motif-based conductance $\Phi(S)$ between them. This motif-based spectral clustering approach has been shown to be very effective in previous studies [7, 40, 60], in which the clique was used as a motif. Similarly, we can use HCS as a motif to devise an HCS-based spectral clustering algorithm by the local counting property of HCSPivot as discussed in Section 4.5. With our HCSPivot, we can get the weight matrix W_M efficiently.

Table 7 presents a comparison of the clustering performance of our proposed HCS-based spectral clustering algorithm against several state-of-the-art graph clustering algorithms, including clique-based (3-clique and 4-clique) spectral clustering [7, 60], Louvain [5], pSCAN [11], label propagation algorithm (LPA) [61], and Infomap [23, 52]. We implement our spectral clustering algorithms using the commonly-used scikit-learn Python package [48]. For other popular algorithms, we use their open-source implementations that have been thoroughly tested [5, 11, 23, 48, 52]. We evaluate the algorithms on the email-Eu-core network downloaded from the SNAP project [36], which contains 42 ground truth communities. Based on the ground truth communities, we can apply 4 widely-used metrics, including ARI [62], Purity [55], NMI [67], and F_1 score, to evaluate the clustering performance of different algorithms. Due to the space limits, the detailed description of these metrics can be found in our full version [39]. As shown in Table 7, our $(3, 1)$-dclique or $(3, 1)$-plex model (when $q = 3$ and $s = 1$, $(3, 1)$-dclique and $(3, 1)$-plex are

Table 7: The results of HCS-based graph clustering

Methods	Metrics			
	ARI	Purity	NMI	F_1
(3, 1)-dclique/plex	**0.47**	**0.67**	**0.68**	**0.49**
(4,1)-dclique	0.34	0.62	0.62	0.38
(4,1)-plex	0.33	0.63	0.63	0.36
3-clique	0.30	0.64	0.64	0.33
4-clique	0.23	0.59	0.59	0.27
Louvain	0.33	0.45	0.58	0.38
pSCAN	0.02	0.28	0.28	0.10
LPA	0.10	0.52	0.49	0.13
Infomap	0.28	0.52	0.62	0.33

identical) achieves the best clustering performance with all 4 metrics. In general, both HCS and clique-based spectral clustering algorithms (the first 5 rows) significantly outperform the other baselines to identify real-world communities, and our HCS-based solutions can further outperform clique-based spectral clustering algorithms [7, 60]. These results demonstrate the high effectiveness of the proposed solutions in community detection applications.

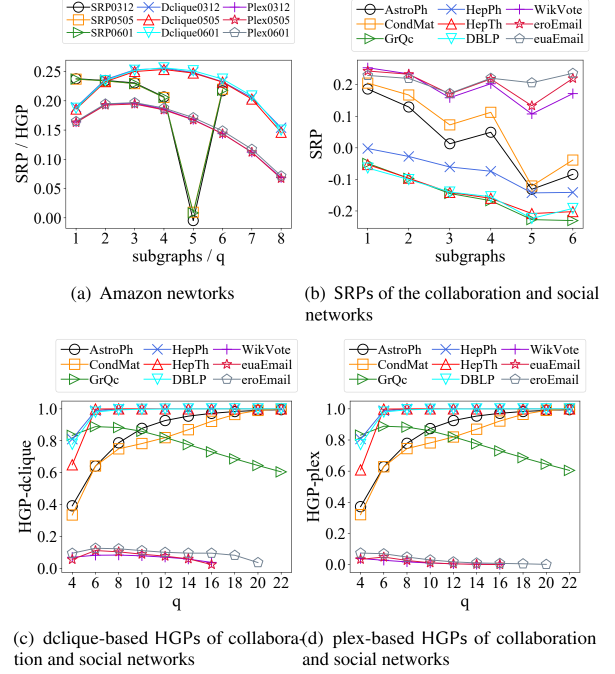
5.4 Application 2 : HCS-based network analysis

The graph profile (GP) is a kind of characteristic vector of given networks, which has a lot of applications in network analysis [3, 29, 43, 44, 68]. A nice property of GP is that the networks in the same domain often have similar GPs [46, 71].

The subgraph ratio profile (SRP) is a well-known and widely used GP [46]. The SRP is a vector that quantifies the relative significance of a sequence of subgraphs, denoted by $\{M_1, M_2, ...\}$, where the i_{th} entry in the vector corresponds to the normalized count of a specific subgraph M_i. This normalization involves subtracting the count of M_i in a randomly generated null graph from that of the original graph, followed by a scaling [46]. In our study, we use a null graph model based on the algorithm proposed in [35], which preserves the core number of the original graph. The sequence of subgraphs $\{M_1, M_2, ...\}$ is composed of six types of connected subgraphs with a size of four, following the methodology of [46].

HCS-based graph profile. We introduce a novel graph profile, termed the HCS-based graph profile (HGP), which is constructed based on the count of HCS. The HGP is a vector where the i_{th} entry represents the ratio between the count of cliques and HCSs with size q_i. Since cliques are a special type of HCS, this ratio can be interpreted as the probability of an HCS being a clique, which characterizes the convergence behavior of the graphs. Since HCSPivot can count HCSs with size in a range $[q_l, q_r]$ simultaneously (Section 4.5), we can compute HGP efficiently.

In Fig. 6, we compare the SRPs and HGPs on the networks in three domains [36]. We set the sequence of size of HGP as $\{4, 5, ..., 20\}$. Fig. 6(a) plots the SRPs and HGPs on 3 Amazon networks. Fig. 6(b), Fig. 6(c) and Fig. 6(d) plot the SRPs, dclique-based HGPs and plex-base HGPs respectively on 6 collaboration and 3 social networks. Similar to SRPs, HGPs of the networks in the same domain have the same change tendency. For example, on the Amazon networks in Fig. 6(a), both Ama0312, Ama0505 and Ama0601 have exactly the same shapes of HGPs and SRPs. The same performance of SRP and HGP implies that the proposed HGP can characterize the network properties in the same domain as SRP. When the networks are in different domains, we find that HGP performs better than SRP. Specifically, in Fig. 6(b), SRP cannot separate the collaboration and social networks. However, in Fig. 6(c) and Fig. 6(d), HGP separates the two kinds of networks clearly. As a result, HGP is a better GP than SRP when there exist networks in multiple domains. These results demonstrate the high effectiveness of the proposed HGP to characterize network properties.



(a) Amazon newtorks
(b) SRPs of the collaboration and social networks
(c) dclique-based HGPs of collaboration and social networks
(d) plex-based HGPs of collaboration and social networks

Figure 6: (1) Fig. 6(a) are SRPs and HGPs on the Amazon networks. The Amazon0312, Amazon0505 and Amazon0601 have the same SRPs and HGPs. (2) Fig. 6(b), Fig. 6(c) and 6(d) are SRPs and HGPs on 6 collaboration (from AstroPh, CondMat to DBLP) and 3 social (from WikVote to eroEmail) networks. In Fig. 6(b), the networks in the two domains have similar SRPs. In Fig. 6(c) and Fig. 6(d), the networks in the same domain still have similar HGPs, but the HGPs of the networks in the two domains are different.

6 RELATED WORK

Subgraph Counting. Our work is related to the subgraph counting problem which includes general subgraph counting and specific subgraph counting [51]. The recent exact algorithms for general subgraphs counting [2, 30, 47, 50] can compute the count efficiently, but they always have a small size constraint. To overcome the hardness, sampling-based approximate algorithms [1, 8, 34, 65] for general subgraph counting are well studied. Recently, machine learning techniques are also applied for subgraph counting [15, 63, 73]. However, these method are often not very accurate and also cannot provide a theoretical guarantee of the estimated counts.

Despite the general algorithms, there are a lot of algorithms designed for counting specific important subgraphs, where the most representative one is k-clique. The first algorithm for k-clique counting was introduced by [16], which is a listing-based solution. To improve the efficiency, some ordering based algorithms [22, 38] are designed. Instead of listing, Jain and Seshadhri [33] propose an elegant algorithm PIVOTER, which can count all k-cliques without listing them, based on the classic maximal clique enumeration algorithm [9]. There also exist several approximate clique counting algorithms [32, 70], which are often more efficient but cannot obtain the exact counts. All these algorithms are tailed to the problem of k-clique counting and cannot be applied to the general HCS counting problem. In addition, we also note that the complexity of the hereditary subgraph counting problem was investigated in [26]. However, no specific counting algorithm was proposed in [26].

Maximal Hereditary Subgraph Enumeration. Our work is also related to the problem of maximal hereditary subgraph enumeration, the goal of which is to enumerate all maximal hereditary subgraphs. The widely studied maximal hereditary subgraph model is the maximal clique. The most popular algorithm for maximal clique enumeration is the classic pivot-based Bron-Kerbosch algorithm [9, 58]. Since clique model is often too restrictive, several relaxed clique models include s-defective clique and s-plex are proposed [14, 20, 21, 28, 57, 72, 75], which also satisfy the hereditary property. Recently, several pivot-based solutions for enumerating maximal s-defective cliques and maximal s-plex were proposed [20, 21, 66], which generalize the classic pivoting technique proposed for maximal clique enumeration [58]. We also note that [17] developed a general algorithm to enumerate all maximal hereditary subgraph based on the reverse search framework [4]. However, this algorithm is often much slower than the pivot-based algorithms when processing real-world graphs. All the above mentioned algorithms are tailored to maximal hereditary subgraph enumeration, and they cannot be used for HCS counting. This is because an HCS can be located in many maximal hereditary subgraphs; and it is very difficult to reduce the repeated counts when using the maximal hereditary subgraphs to count HCSs.

7 CONCLUSION

In this work, we address a new problem of counting hereditary cohesive subgraphs (HCSs) in a graph. To tackle this problem, we first propose a listing-based framework with several carefully-designed pruning techniques to count the HCSs. To further improve the efficiency, we then develop a novel pivot-based framework which counts most HCSs in a combinatorial manner without the need for exhaustive listing. Based on our two frameworks, we devise several specific algorithms to count the s-defective cliques and s-plexes in a graph, which are two relaxed clique models and also satisfy the hereditary property. We conduct comprehensive experiments on 8 real-life networks to evaluate the efficiency of our algorithms. The results reveal that the pivot-based solutions exhibit significantly higher performance, being up to 7 orders of magnitude faster than the listing-based solution. In addition, we also evaluate the performance of our methods in graph clustering and network characterization applications, and the results demonstrate the high effectiveness of the proposed solutions.

REFERENCES

[1] Nesreen K. Ahmed, Nick G. Duffield, Theodore L. Willke, and Ryan A. Rossi. 2017. On Sampling from Massive Graph Streams. *Proc. VLDB Endow.* 10, 11 (2017), 1430–1441.

[2] Nesreen K. Ahmed, Jennifer Neville, Ryan A. Rossi, and Nick G. Duffield. 2015. Efficient Graphlet Counting for Large Networks. In *ICDM*. 1–10.

[3] Uri Alon. 2006. *An introduction to systems biology: design principles of biological circuits*. Chapman and Hall/CRC.

[4] David Avis and Komei Fukuda. 1996. Reverse Search for Enumeration. *Discret. Appl. Math.* 65, 1-3 (1996), 21–46.

[5] Thomas Aynaud. 2020. python-louvain x.y: Louvain algorithm for community detection. `https://github.com/taynaud/python-louvain`.

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

[7] Austin R Benson, David F Gleich, and Jure Leskovec. 2016. Higher-order organization of complex networks. *Science* 353, 6295 (2016), 163–166.

[8] Marco Bressan, Flavio Chierichetti, Ravi Kumar, Stefano Leucci, and Alessandro Panconesi. 2018. Motif Counting Beyond Five Nodes. *ACM Trans. Knowl. Discov. Data* 12, 4 (2018), 48:1–48:25.

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

[10] Mauro Brunato, Holger H. Hoos, and Roberto Battiti. 2007. On Effectively Finding Maximal Quasi-cliques in Graphs. In *LION (Lecture Notes in Computer Science, Vol. 5313)*. 41–55.

[11] Lijun Chang, Wei Li, Lu Qin, Wenjie Zhang, and Shiyu Yang. 2017. pSCAN: Fast and Exact Structural Graph Clustering. *IEEE Trans. Knowl. Data Eng.* 29, 2 (2017), 387–401.

[12] Lijun Chang and Lu Qin. 2018. *Cohesive Subgraph Computation Over Large Sparse Graphs*. Springer Cham.

[13] Lijun Chang, Jeffrey Xu Yu, Lu Qin, Xuemin Lin, Chengfei Liu, and Weifa Liang. 2013. Efficiently computing k-edge connected components via graph decomposition. In *SIGMOD*. 205–216.

[14] Xiaoyu Chen, Yi Zhou, Jin-Kao Hao, and Mingyu Xiao. 2021. Computing maximum k-defective cliques in massive graphs. *Comput. Oper. Res.* 127 (2021), 105131.

[15] Zhengdao Chen, Lei Chen, Soledad Villar, and Joan Bruna. 2020. Can Graph Neural Networks Count Substructures?. In *NeurIPS*.

[16] Norishige Chiba and Takao Nishizeki. 1985. Arboricity and Subgraph Listing Algorithms. *SIAM J. Comput.* 14, 1 (1985), 210–223.

[17] Sara Cohen, Benny Kimelfeld, and Yehoshua Sagiv. 2008. Generating all maximal induced subgraphs for hereditary and connected-hereditary graph properties. *J. Comput. Syst. Sci.* 74, 7 (2008), 1147–1159.

[18] Seshadhri Comandur and Srikanta Tirthapura. 2019. Scalable Subgraph Counting: The Methods Behind The Madness. In *WWW*. 1317–1318.

[19] 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*. 1272–1281.

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

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

[22] Maximilien Danisch, Oana Balalau, and Mauro Sozio. 2018. Listing k-cliques in Sparse Real-World Graphs. In *WWW*. 589–598.

[23] Daniel Edler, Anton Holmgren, and Martin Rosvall. 2023. The MapEquation software package. https://mapequation.org.

[24] Alessandro Epasto, Silvio Lattanzi, Vahab S. Mirrokni, Ismail Sebe, Ahmed Taei, and Sunita Verma. 2015. Ego-net Community Mining Applied to Friend Suggestion. *Proc. VLDB Endow.* 9, 4 (2015), 324–335.

[25] David Eppstein, Maarten Löffler, and Darren Strash. 2013. Listing All Maximal Cliques in Large Sparse Real-World Graphs. *ACM J. Exp. Algorithmics* 18 (2013).

[26] Jacob Focke and Marc Roth. 2022. Counting small induced subgraphs with hereditary properties. In *STOC*. 1543–1551.

[27] Santo Fortunato. 2010. Community detection in graphs. *Physics reports* 486, 3-5 (2010), 75–174.

[28] Jian Gao, Zhenghang Xu, Ruizhi Li, and Minghao Yin. 2022. An Exact Algorithm with New Upper Bounds for the Maximum k-Defective Clique Problem in Massive Sparse Graphs. In *AAAI*. 10174–10183.

[29] Roger Guimera, Marta Sales-Pardo, and Luis AN Amaral. 2007. Classes of complex networks defined by role-to-role connectivity profiles. *Nature physics* 3, 1 (2007), 63–69.

[30] Tomaz Hocevar and Janez Demsar. 2014. A combinatorial approach to graphlet counting. *Bioinform.* 30, 4 (2014), 559–565.

[31] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying k-truss community in large and dynamic graphs. In *SIGMOD*. 1311–1322.

[32] Shweta Jain and C. Seshadhri. 2017. A Fast and Provable Method for Estimating Clique Counts Using Turán's Theorem. In *WWW*. 441–449.

[33] Shweta Jain and C. Seshadhri. 2020. The Power of Pivoting for Exact Clique Counting. In *WSDM*. 268–276.

[34] Madhav Jha, C. Seshadhri, and Ali Pinar. 2015. Path Sampling: A Fast and Provable Method for Estimating 4-Vertex Subgraph Counts. In *WWW*. 495–505.

[35] Katherine Van Koevering, Austin R. Benson, and Jon M. Kleinberg. 2021. Random Graphs with Prescribed K-Core Sequences: A New Null Model for Network Analysis. In *WWW '21*, Jure Leskovec, Marko Grobelnik, Marc Najork, Jie Tang, and Leila Zia (Eds.). 367–378.

[36] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[37] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. 2008. Statistical properties of community structure in large social and information networks. In *WWW*. 695–704.

[38] Ronghua Li, Sen Gao, Lu Qin, Guoren Wang, Weihua Yang, and Jeffrey Xu Yu. 2020. Ordering Heuristics for k-clique Listing. *Proc. VLDB Endow.* 13, 11 (2020), 2536–2548.

[39] Rong-Hua Li, Xiaowei Ye, Fusheng Jin, Yu-Ping Wang, Ye Yuan, and Guoren Wang. 2024. Counting Cohesive Subgraphs with Hereditary Properties. In *Full version:*. https://github.com/LightWant/HCS

[40] Zhenqi Lu, Johan Wahlström, and Arye Nehorai. 2018. Community detection in complex networks via clique conductance. *Scientific reports* 8, 1 (2018), 5982.

[41] R Duncan Luce. 1950. Connectivity and generalized cliques in sociometric group structure. *Psychometrika* 15, 2 (1950), 169–190.

[42] Silvano Martello, David Pisinger, and Paolo Toth. 1999. Dynamic Programming and Strong Bounds for the 0-1 Knapsack Problem. *Management Science* 45, 3 (1999), 414–424.

[43] L-E Martinet, MA Kramer, W Viles, LN Perkins, E Spencer, CJ Chu, SS Cash, and ED Kolaczyk. 2020. Robust dynamic community detection with applications to human brain functional networks. *Nature communications* 11, 1 (2020), 2785.

[44] Oliver Mason and Mark Verwoerd. 2007. Graph theory and networks in biology. *IET systems biology* 1, 2 (2007), 89–119.

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

[46] Ron Milo, Shalev Itzkovitz, Nadav Kashtan, Reuven Levitt, Shai Shen-Orr, Inbal Ayzenshtat, Michal Sheffer, and Uri Alon. 2004. Superfamilies of evolved and

designed networks. *Science* 303, 5663 (2004), 1538–1542.
[47] Mark Ortmann and Ulrik Brandes. 2017. Efficient orbit-aware triad and quad census in directed and undirected graphs. *Appl. Netw. Sci.* 2 (2017), 13.
[48] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.
[49] Jian Pei, Daxin Jiang, and Aidong Zhang. 2005. On mining cross-graph quasi-cliques. In *KDD*. 228–238.
[50] Ali Pinar, C. Seshadhri, and Vaidyanathan Vishal. 2017. ESCAPE: Efficiently Counting All 5-Vertex Subgraphs. In *WWW*. 1431–1440.
[51] Pedro Ribeiro, Pedro Paredes, Miguel E. P. Silva, David Aparício, and Fernando M. A. Silva. 2022. A Survey on Subgraph Counting: Concepts, Algorithms, and Applications to Network Motifs and Graphlets. *ACM Comput. Surv.* 54, 2 (2022), 28:1–28:36.
[52] Martin Rosvall, Daniel Axelsson, and Carl T Bergstrom. 2009. The map equation. *The European Physical Journal Special Topics* 178, 1 (2009), 13–23.
[53] Ahmet Erdem Sariyüce, C. Seshadhri, Ali Pinar, and Ümit V. Çatalyürek. 2015. Finding the Hierarchy of Dense Subgraphs using Nucleus Decompositions. In *WWW*. 927–937.
[54] Satu Elisa Schaeffer. 2007. Graph clustering. *Computer science review* 1, 1 (2007), 27–64.
[55] Hinrich Schutze, Christopher D Manning, and Prabhakar Raghavan. 2008. *Introduction to information retrieval*. Cambridge University Press.
[56] Stephen B Seidman. 1983. Network structure and minimum degree. *Social networks* 5, 3 (1983), 269–287.
[57] 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.
[58] Etsuji Tomita, Akira Tanaka, and Haruhisa Takahashi. 2006. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theor. Comput. Sci.* 363, 1 (2006), 28–42.
[59] Charalampos E. Tsourakakis. 2015. The K-clique Densest Subgraph Problem. In *WWW*. 1122–1132.
[60] Charalampos E. Tsourakakis, Jakub Pachocki, and Michael Mitzenmacher. 2017. Scalable Motif-aware Graph Clustering. In *WWW*. 1451–1460.
[61] Réka Albert Usha Nandini Raghavan and Soundar Kumara. 2007. Near linear time algorithm to detect community structures in large-scale networks. *Phys. Rev. E* 76, 036106 (2007).
[62] Nguyen Xuan Vinh, Julien Epps, and James Bailey. 2009. Information theoretic measures for clusterings comparison: is a correction for chance necessary?. In *Proceedings of the 26th annual international conference on machine learning*. 1073–1080.
[63] Hanchen Wang, Rong Hu, Ying Zhang, Lu Qin, Wei Wang, and Wenjie Zhang. 2022. Neural Subgraph Counting with Wasserstein Estimator. In *SIGMOD*. 160–175.
[64] Jia Wang and James Cheng. 2012. Truss Decomposition in Massive Networks. *Proc. VLDB Endow.* 5, 9 (2012), 812–823.
[65] Pinghui Wang, Junzhou Zhao, Xiangliang Zhang, Zhenguo Li, Jiefeng Cheng, John C. S. Lui, Don Towsley, Jing Tao, and Xiaohong Guan. 2018. MOSS-5: A Fast Method of Approximating Counts of 5-Node Graphlets in Large Graphs. *IEEE Trans. Knowl. Data Eng.* 30, 1 (2018), 73–86.
[66] Zhengren Wang, Yi Zhou, Mingyu Xiao, and Bakhadyr Khoussainov. 2022. Listing Maximal k-Plexes in Large Real-World Graphs. In *WWW*. 1517–1527.
[67] Ian H Witten, Eibe Frank, Mark A Hall, Christopher J Pal, and MINING DATA. 2005. Practical machine learning tools and techniques. In *Data Mining*, Vol. 2.
[68] Xiaoke Xu, Jie Zhang, and Michael Small. 2008. Superfamily phenomena and motifs of networks induced from time series. *Proceedings of the National Academy of Sciences* 105, 50 (2008), 19601–19605.
[69] Jaewon Yang and Jure Leskovec. 2012. Defining and evaluating network communities based on ground-truth. In *KDD*. 1–8.
[70] Xiaowei Ye, Rong-Hua Li, Qiangqiang Dai, Hongzhi Chen, and Guoren Wang. 2022. Lightning Fast and Space Efficient k-clique Counting. In *WWW*. 1191–1202.
[71] Hao Yin, Austin R. Benson, and Jure Leskovec. 2017. Higher-order clustering in networks. *Physical Review E* 97, 5 (2017), 052306.
[72] Haiyuan Yu, Alberto Paccanaro, Valery Trifonov, and Mark Gerstein. 2006. Predicting interactions in protein networks by completing defective cliques. *Bioinform.* 22, 7 (2006), 823–829.
[73] Kangfei Zhao, Jeffrey Xu Yu, Hao Zhang, Qiyan Li, and Yu Rong. 2021. A Learned Sketch for Subgraph Counting. In *SIGMOD*. 2142–2155.
[74] Rui Zhou, Chengfei Liu, Jeffrey Xu Yu, Weifa Liang, Baichen Chen, and Jianxin Li. 2012. Finding maximal k-edge-connected subgraphs from a large graph. In *EDBT*. 480–491.
[75] 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*. 12453–12460.