Nucleus Decomposition Revisited: An Efficient Counting-Based Approach

WENQIAN ZHANG, The University of New South Wales, Australia
ZHENGYI YANG*, The University of New South Wales, Australia
DONG WEN, The University of New South Wales, Australia
YI DING, The University of New South Wales, Australia
WENJIE ZHANG, The University of New South Wales, Australia
XUEMIN LIN, Shanghai Jiao Tong University, China

Nucleus decomposition provides a unified framework for discovering hierarchically cohesive substructures in graphs by generalizing the notions of k-core and k-truss to higher-order (r, s)-nucleus. Existing algorithms suffer from a fundamental bottleneck: they rely on explicit enumeration of all s-cliques, whose number grows combinatorially with s. In this paper, we revisit the existing nucleus decomposition framework and present the first *counting-based* approach that eliminates explicit s-clique enumeration. We propose the *Clique Path Index*, a compact auxiliary structure that encodes the s-clique search space as concise paths, enabling direct computation, efficient dynamic updates, and connectivity maintenance during nucleus decomposition. Extensive experiments on real-world datasets demonstrate that our approach achieves an average speedup of *one order of magnitude* and up to *two orders of magnitude* for both nucleus decomposition and hierarchy construction. More importantly, while existing algorithms often time out even for small values of s, our method scales efficiently to larger s and denser graphs.

CCS Concepts: • **Theory of computation** → **Graph algorithms analysis**; • **Information systems** → *Graph-based database models.*

Additional Key Words and Phrases: nucleus decomposition, clique counting, cohesive subgraphs, hierarchy

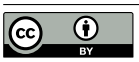
ACM Reference Format:
Wenqian Zhang, Zhengyi Yang, Dong Wen, Yi Ding, Wenjie Zhang, and Xuemin Lin. 2026. Nucleus Decomposition Revisited: An Efficient Counting-Based Approach. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 215 (June 2026), 26 pages. https://doi.org/10.1145/3802092

1 Introduction

Graphs are fundamental data structures consisting of vertices and edges that model relational information. They are widely used across diverse domains, including social network analysis, bioinformatics, and knowledge graph modeling. Beyond pairwise relations, higher-order structures such as cliques and nuclei often provide deeper insights into the underlying organization of networks, revealing cohesive communities and latent functional modules [5, 35]. In social networks, cliques capture tightly connected friendship groups [26, 53]; in biological systems, dense patterns

*Zhengyi Yang is the corresponding author.

Authors' Contact Information: Wenqian Zhang, The University of New South Wales, Sydney, Australia, wenqian.zhang@unsw.edu.au; Zhengyi Yang, The University of New South Wales, Sydney, Australia, zhengyi.yang@unsw.edu.au; Dong Wen, The University of New South Wales, Sydney, Australia, dong.wen@unsw.edu.au; Yi Ding, The University of New South Wales, Sydney, Australia, yi.k.ding@unsw.edu.au; Wenjie Zhang, The University of New South Wales, Sydney, Australia, wenjie.zhang@unsw.edu.au; Xuemin Lin, Shanghai Jiao Tong University, Shanghai, China, xuemin.lin@sjtu.edu.cn.



ACM 2836-6573/2026/6-ART215
https://doi.org/10.1145/3802092

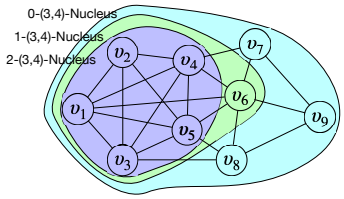


Fig. 1. An example of a graph G.

correspond to protein-protein interactions and functional complexes [5]; in collaboration networks, they represent research groups formed through co-authorship [16, 47]; and on the World Wide Web, hyperlinks induce dense subgraphs corresponding to related content or topical communities [14, 31].

Decomposing graphs into hierarchically cohesive substructures is a fundamental task in graph mining to understand complex network, enabling diverse applications such as community detection [15], influence propagation [20], anomaly detection [2], graph visualization [37], and protein modeling [34]. Common graph decomposition models include the k-core [4, 30] and k-truss [12, 67], which recursively peel away low-degree vertices or edges to expose progressively denser subgraphs. To better capture higher-order dense structures, *nucleus decomposition* was introduced [41], providing finer granularity that uncovers progressively denser and more cohesive subgraphs in real-world networks. It generalizes classical models such as k-core and k-truss by defining cohesiveness in terms of r-cliques and their participation in larger s-cliques, and can be naturally extended to other higher-order structures.

Specifically, given two integers $0 < r < s$, an (r, s)-nucleus is defined as a maximal subgraph in which every r-clique participates in at least k distinct s-cliques (and all such r-cliques are s-connected). The nucleus core number $\kappa_s(R)$ of an r-clique R is defined as the maximum k such that R belongs to a k-(r, s)-nucleus. This generalization subsumes classic notions: the $(1, 2)$-nucleus corresponds to the k-core [43], the $(2, 3)$-nucleus corresponds to the k-truss [66], while higher-order (r, s) values yield clique-based, richer structural representations of network cohesiveness.

Example 1.1. Figure 1 illustrates a sample graph G. Taking the $(3, 4)$-nucleus as an example, the subgraph induced by vertices v_1, v_2, v_3, v_4, and v_5 forms a 2-$(3, 4)$-nucleus, as each 3-clique is contained in at least two 4-cliques. For instance, the 3-clique (v_1, v_2, v_3) belongs to the 4-cliques (v_1, v_2, v_3, v_4) and (v_1, v_2, v_3, v_5). Similarly, all other 3-cliques in this subgraph satisfy this condition. The maximum nucleus core number for the $(3, 4)$-nucleus in this graph is 2.

Applications. Compared with coherent subgraph models such as the k-core and k-truss, the (r, s)-nucleus provides a more general higher-order model, where (r, s) control cohesiveness. In particular, r specifies the base clique unit, while s specifies the cohesiveness of the resulting nuclei and thus how tightly connected they are. A larger s imposes a stricter support requirement: an r-clique must participate in many distinct s-cliques, so nuclei persist only when higher-order overlap is abundant rather than accidental, providing stronger evidence of cohesion [41]. This requirement arises naturally in many domains; we discuss representative examples below:
Coordinated fraud / collusion. From interaction logs, we can build a graph where suspicious accounts tend to act in lockstep on many targets. High-s nuclei help isolate small coordinated rings by filtering out casual triangles and keeping only groups with repeated multi-way coordination [6, 17].
Recommendation / e-commerce. In an item-to-item graph from co-purchase or co-click signals, high-s nuclei highlight compact bundles of items that co-occur again and again across many users, which can be used as stable and interpretable bundles for browsing, diversification, or cold-start organization [27].

Collaboration analysis. In a coauthor graph, high-s nuclei capture stable teams whose collaborations are supported by many different higher-order coauthorship cliques, instead of being connected only through one large paper or a single hub author [33].

Existing Methods and Key Limitation. Despite its effectiveness and broad applicability, computing nucleus decomposition is computationally intensive, motivating extensive research on improving its efficiency [32, 40–42, 44, 45]. Existing studies primarily focus on optimizing the peeling process [44, 45], where the key idea is to iteratively remove low-support r-cliques those that participate in fewer s-cliques to expose progressively denser substructures. Other lines of research explore parallel acceleration, either through local update strategies [40] or multi-threaded peeling algorithms [44, 45].

However, all existing algorithms share a fundamental limitation: they rely on explicit ***clique enumeration***, requiring all s-cliques in the graph to be enumerated. Such enumeration significantly limits scalability, as the number of s-cliques increases combinatorially with s, often reaching trillion scale even for moderate values of s. As a result, existing approaches are typically limited to small parameter settings (e.g., $(3, 4)$-nucleus) or to small and sparse graphs. In theory, the best existing methods achieve asymptotic work complexity comparable to full s-clique enumeration [44, 45], leaving the intrinsic computational barrier of explicit clique listing unresolved, particularly for larger graphs and higher parameter values.

Example 1.2. The number of k-cliques grows explosively with increasing k. We observe that in DBLP, the number of 6-cliques already exceeds 4×10^9 and reaches 1.5×10^{36} at k=57; in WEB-STANFORD, it includes 2.2×10^{12} 28-cliques. Because existing frameworks explicitly enumerate s-cliques, the computation becomes quickly infeasible—even enumerating all 6-cliques alone incurs substantial overhead(Figure 12).

Research Question. This leads to the critical research question and the main objective of this paper: ***How to compute*** (r, s)***-nucleus decomposition without explicitly enumerating*** s***-cliques?***

Challenges. Avoiding s-clique enumeration in nucleus decomposition is highly challenging, as enumeration is deeply embedded in all existing algorithmic frameworks. All existing algorithms rely on explicitly enumerating all s-cliques for three essential reasons:

Computing the support of each r-clique. The support of an r-clique R is defined as the number of s-cliques containing it. Existing methods compute this by enumerating all s-cliques and recording, for each r-clique, the set of s-cliques that include it.

Updating neighboring r-cliques during peeling. When an r-clique R is removed during the peeling process (along with all s-cliques containing it), every r-clique R' that shares an s-clique with R must have its support decremented by 1. This update requires knowing, for each s-clique, the set of r-cliques it contains; thus, adjacency among r-cliques is derived from enumerated s-cliques.

Defining connectivity for hierarchy construction. Two r-cliques R and R' are *s-connected* if a sequence of r-cliques exists such that each consecutive pair of cliques shares at least one common s-clique. To identify nucleus components for hierarchy construction, existing algorithms enumerate and traverse all s-cliques to determine these connectivity relationships.

Our Idea. To efficiently support the above core operations of nucleus decomposition while avoiding explicit s-clique enumeration, we design a novel auxiliary data structure, the *Clique Path Index* (CPI), CPI encodes the space of s-cliques into compact paths, enabling counting, updates, and connectivity queries to be executed directly over these paths without enumeration. The proposed index provides the following key features:

Counting without enumeration. CPI enables s-clique counting and the computation of r-clique supports by aggregating path statistics, thereby avoiding the exponential overhead of enumerating all s-cliques as in existing algorithms.
Dynamic and efficient updates. CPI performs efficient incremental updates during peeling by dynamically modifying or splitting affected clique paths upon deletions. As clique information is aggregated in the path representation, removing a single r-clique implicitly accounts for the removal of many s-cliques, thereby significantly reducing redundant computation during peeling.
Built-in connectivity maintenance. CPI maintains s-connectivity naturally among r-cliques during updates. Connectivity relationships are preserved and adjusted within the indexed paths, enabling hierarchy construction to be performed directly without additional traversal or re-enumeration.
Contributions. Building on the proposed *Clique Path Index* (CPI), we revisit existing enumeration-based algorithms and present a novel ***Counting-Based Nucleus Decomposition*** (CND) framework. Our approach eliminates the need for explicit s-clique enumeration, delivering both theoretical advances and substantial performance improvements for nucleus decomposition on large graphs. It scales efficiently to higher values of s, yields denser and more cohesive substructures, and produces a finer-grained and more informative hierarchical organization. The main contributions of this work are summarized as follows:
Counting-based framework for nucleus decomposition. We propose the *first* counting-based nucleus decomposition framework, which computes nucleus structures through aggregated counting without explicit s-clique enumeration, enabling efficient decomposition on large and dense graphs.
Efficient index for clique counting and maintenance. We design an efficient update mechanism for the proposed CPI during the peeling process. The index compactly encodes s-clique information as paths. The update procedure is further optimized for different sizes of r-clique, including r=1, r=2, r=3, and higher.
Integrated approach for fast hierarchy construction. Our method naturally preserves s-connectivity among r-cliques within the proposed CPI, enabling hierarchy construction to be performed in an integrated phase with minimal overhead and without extra traversal or re-enumeration.
Comprehensive experimental study on real-world graphs. Extensive experiments show that our approach significantly outperforms existing algorithms, achieving an average speedup of *one order of magnitude* and up to *two orders of magnitude* for both nucleus decomposition and hierarchy construction. More importantly, while existing algorithms often time out even for small values of s, our method scales efficiently to larger s and denser graphs.

2 Preliminaries

Graph Notations. We focus on a *simple*, *unweighted* graph, defined as a pair $G = (V, E)$ where V is the set of vertices and $E \subseteq V \times V$ is the set of undirected edges. We use $N_G(v) = \{u \in V \mid (u, v) \in E\}$ to denote the set of neighbors of a vertex $v \in V$ in G, and $\deg_G(v) = |N_G(v)|$ to denote its degree. We use $G[V']$ to denote the subgraph *induced* by a vertex subset $V' \subseteq V$. We store a graph G in the classical Compressed Sparse Row (CSR) format, consisting of two integer arrays: `offsets` of length $|V|$+1 and `edges` of length $|E|$.
Definition 2.1 (k-Clique)**.** A *k-clique* is a vertex set $C \subseteq V$ with $|C| = k$ such that $(u, v) \in E$ for every $u, v \in C$.

Definition 2.2 ((r, s)-Clique Degree)**.** Given two integers $0 < r < s$, the *(r, s)-clique degree* of an r-clique R in G is defined as

$$\deg^G_{(s)}(R) \;=\; \big|\{\, S \subseteq V \mid R \subseteq S,\ S \text{ is a } s\text{-clique in } G \,\}\big|.$$

We also refer to $\deg^{G}_{(s)}(R)$ as the *s-clique support* of R in G. When the graph G is clear from the context, we simply write $\deg_{(s)}(R)$.

Definition 2.3 (k-admissible s-clique)**.** Given integers $0 < r < s$ and $k > 0$, an induced s-clique S in G is called *k-admissible* if every r-clique $R \subseteq S$ satisfies $\deg_{(s)}(R) \geq k$.

Definition 2.4 (k-(r,s)-nucleus)**.** Let $\mathcal{S}_k$ be the set of all k-admissible s-cliques of G. For any subset $\mathcal{S} \subseteq \mathcal{S}_k$, denote $G[\mathcal{S}] = G\left[\bigcup_{S\in\mathcal{S}} S\right]$. A subgraph $G_k = G[\mathcal{S}]$ is a *k-(r,s)-nucleus* if

(1) G_k is *s-connected*: for any two r-cliques $R, R' \subseteq V(G_k)$, there is a sequence $R = R_1, \ldots, R_t = R'$ with $R_i \cup R_{i+1} \subseteq S_i$ for some $S_i \in \mathcal{S}$;
(2) G_k is *maximal*, i.e., no proper superset $\mathcal{S}' \supset \mathcal{S}$ still satisfies condition (1).

A k-(r,s)-nucleus is a maximal subgraph in which every r-clique participates in at least k distinct s-cliques, and all such r-cliques are tightly connected through shared s-cliques.

Definition 2.5 (Core Number)**.** Given two integers $0 < r < s$, the *(r,s)-nucleus core number* (or *coreness*) of an r-clique R, denoted by $\kappa_s(R)$, is the largest integer k such that there exists a k-(r,s)-nucleus G_k in G containing R. Equivalently, it is the largest integer k for which R belongs to some k-(r,s)-nucleus G_k in G.

Problem statement. Given a graph G and integers $0 < r < s$, the goal is to compute the (r,s)-nucleus core number $\kappa_s(R)$ for every r-clique R in G. (The hierarchy construction of k-(r,s) nuclei will be formally discussed in Section 5.)

Computing these core numbers requires tracking, for each r-clique R, its *active* s-clique support $\deg_{(s)}(R)$ (Definition 2.2) under a sequence of deletions. A widely used approach in prior work is the peeling-based nucleus decomposition [39, 41, 42, 44, 45]: starting from the full graph, it iteratively removes the active r-cliques with the smallest current support and updates the supports of the remaining r-cliques. We summarize this baseline next to fix terminology and to expose the scalability bottleneck that motivates our index-based design.

Baseline: Peeling-Based Nucleus Decomposition. Algorithm 1 illustrates the peeling paradigm. In the *initial counting* phase, all r-cliques and s-cliques are enumerated and marked active, and each active r-clique R is assigned its initial support $\deg_{(s)}(R)$. In the peeling loop, the algorithm repeatedly removes the active r-cliques with the smallest support; then every active s-clique that contains a removed r-clique is marked inactive, which decreases the support of the remaining r-cliques that it contains. The coreness $\kappa_s(R)$ of each r-clique is the peeling level at which it is removed.

This framework runs in $O(|K_r| + |K_s|)$ time [44], where $|K_r|$ and $|K_s|$ denote the numbers of r-cliques and s-cliques, respectively. However, both time and space critically depend on *explicitly enumerating* s-cliques (during initial counting and during support updates). Since $|K_s|$ can grow combinatorially with s, this dependence quickly becomes prohibitive as s increases, which explains why prior methods typically target only small configurations (e.g., $(4,5)$-nucleus) on small or sparse graphs. Local update-based variants have been proposed to improve parallelism [40], but they still rely on s-clique enumeration and thus inherit the same bottleneck. These limitations motivate algorithmic designs that avoid explicit s-clique enumeration.

3 The Counting-Based Framework

In this section, we present our core idea for nucleus decomposition, which addresses the limitations of existing methods by eliminating their reliance on explicit s-clique enumeration. The key idea of our algorithm is to leverage the proposed *Clique Path Index* (CPI) to directly count s-cliques while maintaining and updating the index dynamically during the peeling process. In the following, we first provide an overview of our counting-based framework, then review the background on clique counting, and finally present the definition and design of CPI.

Algorithm 1: Peeling Framework

Input: Graph G, clique sizes $0 < r < s$
Output: $\kappa_s(R)$ for every r-clique R

Enumerate all r-cliques in G;
Enumerate all s-cliques in G;
mark every s-clique $S \in G$ as `active`;
$\delta(R) \leftarrow$ the number of s-cliques that contain R
$coreness \leftarrow 0$
while *there exists* `active` *r-cliques* **do**
  find $c = \min\{\delta(R) \mid R$ is `active`$\}$
  $coreness \leftarrow max(c, coreness)$
  let $\mathcal{R} = \{R \mid \delta(R) = c,\ R$ `active`$\}$
  foreach $R \in \mathcal{R}$ **do**
    $\kappa_s(R) \leftarrow coreness$
    mark R as `inactive`;
    foreach `active` *s-clique* S *with* $R \subseteq S$ **do**
      mark S as `inactive`;
      foreach `active` *r-clique* $R' \subseteq S$ **do**
        $\delta(R') \leftarrow \delta(R') - 1$;
return $\kappa_s(R)$ *for every r-clique* $R \in G$

Algorithm 2: (r, s)-CND

Input: Graph $G = (V, E)$, clique sizes $r < s$
Output: Core value $\kappa_s(R)$ for every r-clique R

$\mathcal{B} \leftarrow$ BuildIndex(G);
`// initialize s-support for all r-cliques`
$support \leftarrow s$-CliqueCount$(\mathcal{B}, r, s)$
build a min-heap $H_{\min}$ using $support$;
$c_{min} \leftarrow 0$;
while $H_{\min}$ *is not empty* **do**
  $R_{min} \leftarrow$ ExtractMin$(H_{\min})$;
  $c_{min} \leftarrow \max(c_{min}, support(R_{min}))$;
  $\kappa_s(R_{min}) \leftarrow c_{min}$;
  UpdateIndex$(R_{min}, \mathcal{B})$; `// remove` R_{min} `from` $\mathcal{B}$
  `// update support of affected r-cliques`
  UpdateSupport$(R_{min}, \mathcal{B}, support, H_{\min})$
return $\kappa_s(R)$ *for all r-cliques* R *in* $\mathcal{B}$;

3.1 Overview

We observe that, during the computation of nucleus core decomposition, a key operation is measuring the support of r-cliques, which intuitively depends only on counting the number of s-cliques containing each r-clique. Hence, we believe avoiding explicit enumeration of s-cliques is both feasible and desirable. Based on this observation, we propose a minimal counting-based framework, as shown in Algorithm 2.

Key Idea. The core idea of our framework is to avoid explicit enumeration of s-cliques through an auxiliary index structure. Our method follows the peeling paradigm but eliminates explicit s-clique enumeration by

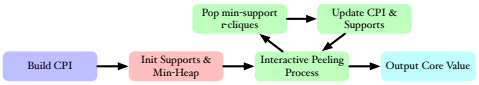


Fig. 2. Overview of our approach.

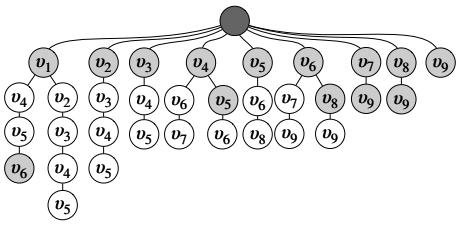


Fig. 3. An example of a Succinct Clique Tree (SCT). White nodes denote pivots; gray nodes denote holds.

introducing four key operations supported by this index: (i) BuildIndex, which initializes the auxiliary index structure (Line 1); (ii) s-CliqueCount, which computes the initial s-support for each r-clique (Line 2); (iii) UpdateIndex, which removes peeled r-cliques from the index (Line 9); and (iv) UpdateSupport, which identifies and updates the supports of affected r-cliques after each removal (Line 10). Figure 2 illustrates the overall pipeline of our approach.

Index Support. With an index that efficiently supports these four operations, nucleus decomposition can be performed without enumerating any s-cliques, making the algorithm independent of enumeration and effectively eliminating the bottleneck.

3.2 Succinct Clique Tree

We first review the state-of-the-art clique counting method, the *Succinct Clique Tree* (SCT) [19]. The SCT compacts the Bron–Kerbosch pivoting search space for exact clique counting. Instead of enumerating all cliques, SCT constructs a hierarchical search tree in which each node corresponds to a state in the Bron–Kerbosch maximal clique search (the pivot-based version) [7, 52], thereby implicitly representing all subcliques. Vertices are processed in *degeneracy order*, and edges are oriented to define $N^+(\cdot)$. For each root vertex v in SCT, every state maintains two sets: V_h (hold, already fixed) and V_p (pivot, eligible extensions). Pivoting on $u \in V_p$ branches only on $V_p \setminus N(u)$ while restricting candidates to $V_p \cap N(\cdot)$. Thus, each *clique* path succinctly represents a family of cliques without enumerating them explicitly. This structure supports aggregated counting of all subcliques (of sizes r, s, etc.), enabling exact clique counting without enumeration.

Figure 3 illustrates the SCT constructed for the example graph shown in Figure 1. We first define a *clique path* as follows:

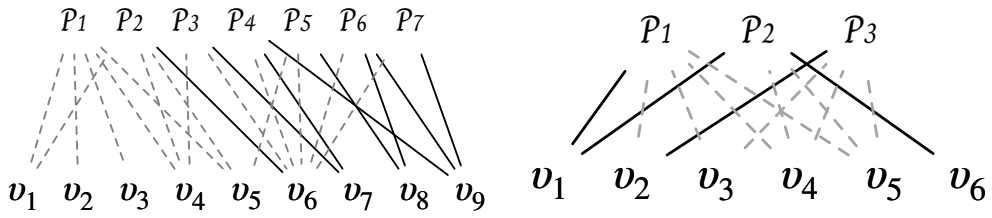
Definition 3.1 (Clique Path)**.** In a Succinct Clique Tree, a *clique path* $\mathcal{P}$ is any root-to-leaf path. All clique paths are enumerated in left-to-right depth-first order, with the i-th path denoted by $\mathcal{P}_i$. Formally, $\mathcal{P}_i = \langle v_{a_1}, v_{a_2}, \ldots, v_{a_\ell} \rangle$, where v_{a_1} is the root, v_{a_ℓ} is the leaf, and ℓ is the path length.

Example 3.1. Figure 3 shows an example of a SCT. $\mathcal{P}_1 = \langle v_1, v_4, v_5, v_6 \rangle$, the second path is $\mathcal{P}_2 = \langle v_1, v_2, v_3, v_4, v_5 \rangle$. There are eleven clique paths in total, and the last path is $\mathcal{P}_{11} = \langle v_8, v_9 \rangle$.

Lemma 3.1 (Unique Path Encoding [19])**.** *For any k-clique C_k in the graph, there exists only one clique path $\mathcal{P} = V_h(\mathcal{P}) \cup V_p(\mathcal{P})$ such that $V_h(\mathcal{P}) \subseteq C_k \subseteq (V_h(\mathcal{P}) \cup V_p(\mathcal{P}))$. Here, $V_h(\mathcal{P})$ and $V_p(\mathcal{P})$ denote the sets of hold vertices and pivot vertices on the path $\mathcal{P}$, respectively.*

Intuitively, any k-clique represented by a path $\mathcal{P}$ must include all hold vertices of that path, while the pivot vertices are optional. Formally, a k-clique C_k is *encoded* by a CPI path $\mathcal{P}$ if and only if $V_h(\mathcal{P}) \subseteq C_k \subseteq V_h(\mathcal{P}) \cup V_p(\mathcal{P})$; it is *covered* by $\mathcal{P}$ if and only if $C_k \subseteq V_h(\mathcal{P}) \cup V_p(\mathcal{P})$.

Example 3.2. For SCT as shown in Figure 3. $V_p(\mathcal{P}_1) = \{v_4, v_5\}$ and $V_h(\mathcal{P}_1) = \{v_1, v_6\}$. Thus, the number of 3-cliques on $\mathcal{P}_1$ is equal to $\binom{2}{3-2} = 2$. $V_p(\mathcal{P}_2) = \{v_2, v_3, v_4, v_5\}$ and $V_h(\mathcal{P}_2) = \{v_1\}$. Thus, the number of 3-cliques on $\mathcal{P}_2$ is equal to $\binom{4}{3-1} = 6$.



(a) An example of a CPI on Figure 1 (no pruning). **(b)** An example of a pruned CPI for $s = 4$ on Figure 3.

Fig. 4. Examples of CPI before and after pruning. Dashed nodes denote pivots, and solid nodes denote holds.

Inadequacies in SCT **Structures.** SCT is designed solely for clique counting on static graphs. For the key functions in our framework (Algorithm 2), SCT can support s-CliqueCount (i.e., initializing the support) with minor modifications. However, it lacks dynamic update capabilities and cannot be trivially extended to support UpdateIndex and UpdateSupport, which are required in our counting-based framework to remove peeled r-cliques efficiently. This limitation motivates the design of a new index that can support all these operations, which is both non-trivial and challenging.

3.3 Clique Path Index

Building on the basic idea of SCT [19], we introduce a new data structure, the *Clique Path Index* (CPI), which is adapted to our counting-and-peeling framework and supports efficient updates. Unlike SCT, CPI represents the search space as independent paths in a bipartite-like layout between vertices and paths, as illustrated in Figure 4a. Thanks to this bipartite-like storage structure, we can also efficiently retrieve the paths containing each vertex and the vertices contained in each path, facilitating the implementation of the UpdateSupport function in our framework to accurately identify affected r-cliques. Updating a traditional tree is difficult, since shared prefixes cause cascaded changes and costly subtree reattachments; accordingly, CPI admits more efficient updates (details in the next section).

Example 3.3. Figure 4a shows an example of a CPI consisting of three paths and five vertices. Path 1 contains vertices v_1, v_2, v_3, v_4, v_5, where v_1 is a hold vertex and the others are pivot vertices. Path 2 contains vertices v_1, v_4, v_5, v_6, where v_1 and v_6 are hold vertices and the others are pivot vertices. Path 3 contains vertices v_2, v_3, v_4, v_5, where v_2 is a hold vertex and the others are pivot vertices.

Based on Lemma 3.1, we derive Lemma 3.2, which enables the computation of the number of s-cliques for each r-clique without enumerating any s-cliques.

Lemma 3.2 (Local s-support of an r-clique)**.** *Let $\mathcal{P}$ be a path in the* CPI *with pivot set V_p and hold set V_h ($h = |V_h|$, $p = |V_p|$), and fix integers $0 < r < s$. For any r-clique $C_r \subseteq \mathcal{P}$ set $b = |C_r \cap V_p|$, Then the number of distinct s-cliques on $\mathcal{P}$ that contain C_r equals $\#_s(C_r) = \binom{p-b}{s-h-b}$.*

Proof. Every s-clique on the path is $V_h \cup Q$ with $Q \subseteq V_p$ and $|Q| = s - h$ (Lemma 3.1). Fix any r-clique C_r and let $b = |C_r \cap V_p|$. Since C_r already contributes those b pivots, an s-clique containing C_r is obtained precisely by choosing the remaining $s - h - b$ pivots from the $p - b$ unused pivots in V_p. The number of such choices is $\binom{p-b}{s-h-b}$, which is zero whenever the lower or upper bound is violated. □

Based on Lemma 3.2, we derive Algorithm 3 to compute the number of s-cliques for each r-clique without explicit enumeration.

Path Pruning. Naively building the index would include information about all cliques in the graph. However, in (r, s)-nucleus decomposition, not all cliques are necessary—we are only interested in cliques that contains s-cliques. To accelerate index construction and reduce its size, we introduce the following pruning rule.

Lemma 3.3 (Feasibility of Path)**.** *Let $\mathcal{P}$ be a path in the* CPI *and let $s \geq 1$ be the target clique size. Then $\mathcal{P}$ encodes at least one s-clique iff $|V_h(\mathcal{P})| \leq s \leq |V_h(\mathcal{P})| + |V_p(\mathcal{P})|$.*

Algorithm 3: s-CliqueCount

Input: Clique Path Index $\mathcal{B}$, clique sizes $0 < r < s$
Output: $count(C_r)$ for every r-clique $C_r \subseteq G$

initialise $count(C_r) \leftarrow 0$ for all r-cliques in $\mathcal{B}$;
foreach *clique path* $\mathcal{P}$ *in* $\mathcal{B}$ **do**
  $V_h \leftarrow \{v \in \mathcal{P} \mid v \text{ is hold}\}$;
  $V_p \leftarrow \{v \in \mathcal{P} \mid v \text{ is pivot}\}$;
  foreach r-*set* $C_r \subseteq \mathcal{P}$ **do**
    $b \leftarrow |C_r \cap V_p|$
    $count(C_r)$ += $\binom{|V_p|-b}{s-|V_h|-b}$;
return $count(C_r)$ *for all* r-*cliques* C_r;

Algorithm 4: Build–CPI(G, s)

Input: Undirected graph $G = (V, E)$, optional size cap s
Output: Clique Path Index (CPI) of G

Compute a degeneracy ordering $(v_1, \ldots, v_n)$ of G Orient every edge from lower to higher rank to obtain $N^+(\cdot)$;
for $i \leftarrow 1$ **to** n **do** // one root per vertex
  $V_{Hold} \leftarrow \{v_i\}$ $P \leftarrow N^+(v_i)$ $V_{Pivot} \leftarrow \emptyset$
  ExpandPath(P, V_{Hold}, V_{Pivot});
Function ExpandPath(P, V_{Hold}, V_{Pivot})**:**
  if $|V_{Hold}| > s$ **then return** ;
  if $P = \emptyset$ **then**
    if $|V_{Hold}| + |V_{Pivot}| \geq s$ **then**
      output path $\{V_{Hold}, V_{Pivot}\}$;
    return
  $u_{piv} \leftarrow \arg\max_{x \in P} |P \cap N^+(x)|$;
  for $v \in P \setminus N(u_{piv})$ **do** // non-neighbors of pivot
    $P \leftarrow P \setminus \{v\}$
    if $v = u_{piv}$ **then**
      ExpandPath($P \cap N(v)$, V_{Hold}, $V_{Pivot} \cup \{v\}$)
    else
      ExpandPath($P \cap N(v)$, $V_{Hold} \cup \{v\}$, V_{Pivot})

Proof. ($\Rightarrow$)According to Lemma 3.1, all s-cliques on $\mathcal{P}$ consist of vertices from $V_h(\mathcal{P})$ and $V_p(\mathcal{P})$. Therefore, if $s < |V_h(\mathcal{P})|$, it is impossible for any s-clique to include all hold vertices, and thus no s-clique exists. Similarly, if $s > |V_h(\mathcal{P})| + |V_p(\mathcal{P})|$, it is impossible for any s-clique to include all hold vertices and enough pivot vertices, and thus no s-clique exists. ($\Leftarrow$)Conversely, if s lies within this interval, we can construct at least one s-clique based on Lemma 3.1, ensuring the existence of at least one s-clique. □

Example 3.4. Considering the CPI shown in Figure 4b, if we need to count 4-cliques, we can prune this CPI to retain only three paths, as illustrated in Figure 4b. Compared to the SCT (Figure 3), the pruned CPI has fewer paths, making it more efficient to store and traverse.

Index Construction. Algorithm 4 illustrates the construction process of CPI. Contrary to SCT, CPI outputs an entire path at once and applies Lemma 3.3 (Line 6 & 9) to prune paths during construction.

Table 1. Instantiation mapping of the unified peeling framework; the CPI/heap data structures are identical across regimes.

Regime	Peeled unit	s-clique support $\deg^{(s)}(\cdot)$	Affected units for local updates
General	r-clique	$\#\{K_s \mid C_r \subseteq K_s\}$	r-cliques on the updated CPI paths
$r = 2$	edge (2-clique)	$\#\{K_s \mid e \subseteq K_s\}$ (*triangle support* when s=3)	units co-occurring with e on the updated CPI paths
$r = 1$	vertex (1-clique)	$\#\{K_s \mid v \subseteq K_s\}$ (*deg* when s=2)	units co-occurring with v on the updated CPI paths

Since paths in the CPI can be pruned to retain only those that must contain s-cliques, we can also derive the following result.

Theorem 3.1 (Number of Clique Paths). *Let $\mathcal{B}$ be the* CPI *of a graph G, and let $\mathcal{P}_s = \{\mathcal{P} \mid |V_h(\mathcal{P})| \leq s \text{ and } |\mathcal{P}| \geq s\}$ be the set of* CPI *paths that remain after pruning for a given integer $s \geq 1$. Let $\mathcal{K}_s$ denote the set of all s-cliques in G. Then $|\mathcal{K}_s| \geq |\mathcal{P}_s|$. Moreover, $\sum_{P \in \mathcal{P}_s} |V_h(P)| \leq s\,|\mathcal{K}_s|$.*

Proof. Let $\mathcal{P}$ be a path in the CPI $\mathcal{B}$. By Lemma 3.3, if $\mathcal{P}$ survives the pruning rule, it contains at least one s-clique. Thus, every path in $\mathcal{P}_s$ contains at least one s-clique. Therefore, the number of s-cliques is greater than or equal to the number of CPI paths in $\mathcal{P}_s$. By definition of $\mathcal{P}_s$, each $P \in \mathcal{P}_s$ satisfies $|V_h(P)| \leq s$. Thus $\sum_{P \in \mathcal{P}_s} |V_h(P)| \leq s|\mathcal{P}_s| \leq s|\mathcal{K}_s|$. □

Remark. This theorem demonstrates a key advantage of our approach. The number of paths in the CPI *cannot exceed the number of s-cliques, ensuring that storing the entire index is no more expensive than storing all s-cliques. More importantly, since each path compactly represents multiple s-cliques, traversing and updating the* CPI *becomes substantially more efficient than operating on enumerated s-cliques. This property underpins the benefits of our counting-based framework, enabling efficient decomposition.*

Complexity. The worst-case time to build the CPI is $O\big(n \cdot 3^{\alpha/3}\big)$ [19], where α is the degeneracy of the graph, $n \cdot 3^{\alpha/3}$ is the maximum number of maximal cliques in a graph with n vertices. But in practice, due to path pruning, the actual construction time is significantly lower than this worst-case bound. Space complexity to store CPI is $O(min(n \cdot 3^{\alpha/3}, |K_s|))$ (Theorem 3.1), where $|K_s|$ is the number of s-cliques in the graph.

From Counting to Decomposition. The CPI provides a powerful tool for computing the number of s-cliques associated with each r-clique while avoiding explicit s-clique enumeration. However, efficiently updating the CPI during computation and leveraging it to guide the decomposition process remain major technical challenges for performing nucleus core decomposition.

4 Updating the Clique Path Index

In this section, we present how to efficiently update CPI during decomposition, specifically how to remove r-cliques (and their corresponding s-cliques) during the peeling process. This operation is one of the most challenging aspects of maintaining CPI, as it must preserve the path semantics defined in Lemma 3.1. We first introduce the general path-splitting principle for arbitrary r, and then show how it simplifies for $r = 2$ and $r = 1$ as implementation optimizations. Throughout peeling, we maintain the same CPI state across all r, only the instantiated update operator differs depending on whether the peeled object is a vertex (r=1), an edge (r=2), or a general r-clique. Table 1 summarizes how the same peeling/update pipeline instantiates for general r, $r = 2$, and $r = 1$.

We first define the semantics of removing an r-clique R from the CPI. After the update, no path in the CPI contains R. Namely, the operation removes exactly those cliques that include R, while preserving all others: every s-clique S with $R \not\subseteq S$ remains encoded by exactly one path in the CPI.

4.1 Our General Update for r-Cliques

In this subsection, we generalize the update strategy from vertices and edges to arbitrary r-cliques.

Conflict Sets and Path Validity. Removing an r-clique R is a higher-order update. It simultaneously affects all s-cliques that contain R. We define the *conflict set* of R as its vertex set $V(R)$. After the update, no clique path may realize all vertices in $V(R)$ at the same time. Meanwhile, every remaining s-clique must still be

Algorithm 5: UpdateIndex

Input: clique path $\mathcal{P} = \{V_H, V_P\}$, set of r-cliques to remove $C_{\text{rm}} = \{C_0, \ldots, C_{t-1}\}$, clique size s
Output: All subpaths induced after deleting C_{rm}

```
Build inverted index I(v) ← { i | v ∈ C_i } for all v ∈ P
cnt← array of size t initialized to r;
PathSplit(V_H, V_P);
Function PathSplit(V_H, V_P)
    if |V_H ∪ V_P| < s or |V_H| > s then return;
    C_j ← the first r-clique in C_rm with cnt[j] = r;
    if no such C_j exists then
        output path {V_H, V_P};
        return
    foreach v ∈ C_j ∩ V_P do
        foreach g ∈ I(v) do
            cnt[g] ← cnt[g] − 1
        V_P ← V_P \ {v};
        PathSplit(V_H, V_P);
        V_H ← V_H ∪ {v};
        foreach g ∈ I(v) do
            cnt[g] ← cnt[g] + 1
```

encoded by exactly one path. On a path, vertices of R can appear as hold or pivot. Holds are mandatory, while pivots represent alternative choices. Therefore, updating the CPI reduces to removing exactly those realizations that include the whole conflict set. If needed, we split the affected path into a small number of new paths to restore validity and uniqueness (Lemma 3.1). Crucially, the number of new paths depends only on how many vertices of R act as pivots on that path. Building on this observation, Theorem 4.1 formalizes the general path-splitting rule for arbitrary r.

Theorem 4.1 (Clique Removal)**.** *Let $\mathcal{P} = (V_p^{\mathcal{P}}, V_h^{\mathcal{P}})$ be a clique path, and let $C = (V_p^C, V_h^C)$ be an r-clique such that $V_p^C \subseteq V_p^{\mathcal{P}}$ and $V_h^C \subseteq V_h^{\mathcal{P}}$. Given an ordered set $V_p^C = \{v_0, \ldots, v_{t-1}\}$, the updated path set $\mathcal{P}'$ is defined as:*

$$\mathcal{P}' = \left\{ \left(V_p^{\mathcal{P}} \setminus \{v_0, \ldots, v_i\},\ V_h^{\mathcal{P}} \cup \{v_0, \ldots, v_{i-1}\}\right) \,\middle|\, i = 0, \ldots, t-1 \right\}.$$

Proof. On a CPI path, holds are mandatory and pivots optional. For each i, promoting $v_0, \ldots, v_{i-1}$ to holds and deleting v_i keeps the path valid and forbids any clique that contains all pivots of C, so no subpath encodes C (soundness). For any surviving clique K, let i be the first index with $v_i \notin K$; then K contains the promoted prefix and excludes v_i, hence is encoded precisely on subpath i and on no other (completeness & uniqueness). □

Example 4.1. Assume we have a clique path $\mathcal{P} = \langle v_1, v_2, v_3, v_4, v_5, v_6 \rangle$, where $V_p = \{v_1, v_2, v_3, v_6\}$ and $V_h = \{v_4, v_5\}$. If we have an r-clique $C = \{v_1, v_2, v_3, v_4\}$, then $V_p^C = \{v_1, v_2, v_3\}$ and $V_h^C = \{v_4\}$. According to Theorem 4.1, we can split $\mathcal{P}$ into three subpaths: $V_{p,0} = \{v_1, v_2, v_6\}, V_{h,0} = \{v_4, v_5\}$, $V_{p,1} = \{v_1, v_6\}, V_{h,1} = \{v_3, v_4, v_5\}$, $V_{p,2} = \{v_6\}, V_{h,2} = \{v_2, v_3, v_4, v_5\}$.

Remark. Theorem 4.1 allows a clique path $\mathcal{P}$ in the CPI *to be split into multiple paths that exclude the r-clique C. This operation is crucial, as it implicitly removes r-cliques—without explicit enumeration—while preserving the correctness of the remaining structure.*

Batched Updates. When multiple cliques are removed simultaneously, they often overlap, leading to redundant updates on shared vertices and paths. To mitigate this overhead, we aggregate overlap ping updates into vertex-centric batches. Specifically, we construct an *inverted index* that maps each vertex to the set of cliques

con taining it. Instead of processing cliques sequentially, we iterate over vertices in this index and update all affected clique paths per ver tex. This vertex-grouped strategy effectively merges overlapping operations and reduce redundant updates.

Algorithm. Algorithm 5 maintains a counter *cnt* to track the number of vertices from each r-clique in C_{rm} that are still present on the current path. When *cnt* equals r, it indicates that the path violates the conflict clique and requires splitting. The splitting process (Function `PathSplit`) recursively handles each r-clique to be removed. First, in Line 6, we prune the path based on Lemma 3.3; if the current path cannot contain any cliques of size s, we return immediately. Then, in Line 7, we select the first r-clique C_j that needs to be removed. If no such r-clique exists, it indicates that the current path does not require any deletion operations, so we report the current path as a subpath (Line 8). Otherwise, we iterate through each vertex v in the r-clique C_j, remove it from the path, and decrement the *cnt* of all r-cliques containing v by 1, as one vertex has been removed from these cliques. In Lines 15-17, we promote all pivot nodes before v to hold nodes according to Theorem 4.1, remove v from the path, and then recursively call the function `PathSplit` to process the updated path. Finally, in lines 19-20, we remove v from the path and continue processing the next vertex. According to Theorem 4.1, we can ensure that after each path split, the resulting subpaths do not contain the removed r-clique, and all size-s cliques that do not contain the removed r-clique are uniquely represented.

Correctness. We prove the correctness of Algorithm 5 in terms of soundness, completeness, and uniqueness. *(Soundness)* The algorithm maintains a counter array *cnt* to track how many vertices of each removed r-clique remain on the current path. A path is emitted only when all *cnt* values are strictly less than r, ensuring that the new path encodes no r-cliques marked for removal. *(Completeness & Uniqueness)* When removing vertices of an r-clique iteratively, at each deletion step, all remaining cliques not containing the removed vertex are re-encoded uniquely in new paths (by Theorem 4.1). Since every vertex is removed exactly once, all valid combinations of remaining vertices are exhaustively enumerated and retained. Therefore, every surviving clique is preserved and encoded by exactly one path in the updated CPI, ensuring both completeness and uniqueness.

Implementation. The size of the residual graph is bounded by the maximum clique size of the input graph, which is typically small for real-world datasets (Table 3). This allows us to represent the residual graph as an adjacency matrix and employ *bitsets* to encode vertex sets on paths, enabling highly efficient set operations (AND, OR, XOR, COUNT) through word-level parallelism [52].

Complexity. Let a clique path $\mathcal{P} = \{V_p(\mathcal{P}), V_h(\mathcal{P})\}$. The worst-case time complexity of Algorithm 5 is $O(\prod_{C \in C_{\text{rm}}} |C \cap V_p(\mathcal{P})|)$, as it may recursively split for each clique in C_{rm}. However, in practice, due to s-pruning and overlap among conflict cliques, the actual number of splits and the running time are significantly lower than this theoretical bound.

4.2 Case $r = 2$

For $r = 2$, our general update framework naturally degenerates to an efficient edge-removal procedure. Specifically, we first derive an *edge-removal theorem* as a direct specialization of our general path-splitting theorem: on any affected clique path, the update reduces to a constant number of cases determined by whether the two endpoints act as hold or pivot, as formalized in Theorem 4.2.

Theorem 4.2 (Edge Removal)**.** *Let $\mathcal{P} = (V_p(\mathcal{P}),\ V_h(\mathcal{P}))$ be a clique path in the* CPI *of G, and let $e = \{u, v\}$ with $u, v \in V(\mathcal{P})$. For any removed e from G, the updated $\mathcal{P}'$ satisfies:*

(1) (hold-hold) If $u, v \in V_h(\mathcal{P})$, then $\mathcal{P}' = \emptyset$;
(2) (hold-pivot) If $u \in V_h(\mathcal{P})$ and $v \in V_p(\mathcal{P})$ (or symmetrically), then $\mathcal{P}' = \left\{ \left(V_p(\mathcal{P}) \setminus \{v\},\ V_h(\mathcal{P})\right) \right\}$;
(3) (pivot-pivot) If $u, v \in V_p(\mathcal{P})$, then

$$\mathcal{P}' = \left\{ \left(V_p(\mathcal{P}) \setminus \{v\},\ V_h(\mathcal{P})\right),\ \left(V_p(\mathcal{P}) \setminus \{u\},\ V_h(\mathcal{P}) \cup \{v\}\right) \right\}.$$

Proof. We prove the theorem separately for three distinct cases. (1) If e is a hold-hold edge, every encoded clique would require both endpoints, which becomes impossible after deleting e; hence, the path is invalid. (2) If e is a hold-pivot edge, all encoded cliques must retain the hold endpoint. Thus, we update the path with respect to the pivot endpoint. (3) If e is a pivot-pivot edge, we split the path into two: (a) $\left(V_p(\mathcal{P}) \setminus \{v\},\ V_h(\mathcal{P})\right)$,

Algorithm 6: UpdateIndex ($r = 2$)

Input: clique path $\mathcal{P} = \{V_H, V_P\}$, remove-edge set E_{rm}, $s > 0$
Output: All subpaths induced after deleting E_{rm}

if $\exists\, e \in E_{\text{rm}} :\ e \subseteq V_h(\mathcal{P})$ **then return**;
foreach $e \in E_{\text{rm}} :\ |e \cap V_h(\mathcal{P})| = 1$ **do**
    remove $(e \cap V_p(\mathcal{P}))$ from $\mathcal{P}$
    remove e from E_{rm}
$H \leftarrow$ a residual graph on $V(\mathcal{P})$ with edges in E_{rm} removed
$v_{fixed} \leftarrow \{\, v \in V(H) \mid v$ is not incident to any edge of $E_{\text{rm}}\,\}$
$Piv \leftarrow \{\, v \in v_{fixed} \mid v$ is a pivot on $\mathcal{P}\,\}$
$R \leftarrow v_{fixed}, \quad P \leftarrow V(H) \setminus v_{fixed}$
`PathUpdate`(R, P, Piv)
Function `PathUpdate`$(R,\ P,\ Piv)$**:**
    if $|R \setminus Piv| > s$ **then return**;
    if $P = \emptyset$ **then**
        if $|R| \le s$ **then output path** $\{R,\ Piv\}$;
        return
    $u_{piv} \leftarrow \arg\max_{x \in P} |P \cap N_H(x)|$;
    foreach $v \in P \setminus N_H(u_{piv})$ **do**
        $R' \leftarrow R \cup \{v\}; \quad P' \leftarrow P \cap N_H(v)$;
        if $v = u_{piv}$ **then** $Piv' \leftarrow Piv \cup \{v\}$;
        else $Piv' \leftarrow Piv$;
        `PathUpdate`$(R',\ P',\ Piv')$;
        $P \leftarrow P \setminus \{v\}$;

which removes all cliques encoded by the path that contain v; (b) $\big(V_p(\mathcal{P}) \setminus \{u\},\ V_h(\mathcal{P}) \cup \{v\}\big)$, which restores all cliques containing v but not u. □

Batched Updates. Theorem 4.2 specifies how a single edge deletion updates one clique path. In peeling, many edges are removed together. Processing them one by one revisits the same paths repeatedly. This is most expensive for pivot–pivot deletions, where each deletion may split a path and trigger cascading updates.

We use a *path-local* batched strategy. For each affected clique path, we first merge all removed edges that touch this path. We then handle hold-hold and hold-pivot deletions immediately (they either delete the path or drop the pivot). For the remaining pivot-pivot deletions, we build a residual graph on $V(P)$ after removing these edges. Finally, we reconstruct the resulting clique paths from the residual graph in a single pass. The residual graph is small (bounded by the maximum clique size), so the reconstruction is efficient.

Key Idea. We first handle the simple cases (hold-hold removals delete the entire path, and hold-pivot removals exclude the pivot), before addressing the more complex scenario of removing multiple pivot-pivot paths simultaneously. For pivot-pivot paths, the core idea is to first construct a *residual graph* from vertices in $V(\mathcal{P})$, retaining only the edges that remain valid within this subgraph. We then reconstruct new clique paths from the residual graph. Since the residual graph is small (no larger than the maximum clique in the graph), this reconstruction can be performed efficiently, avoiding redundant computation.

Algorithm. The algorithm (Algorithm 6) takes as input a clique path $\mathcal{P}$, a set E_{rm} of edges to remove, and the target clique size s (for pruning). It first constructs the residual graph H by deleting all edges in E_{rm} from the complete graph induced by $V(\mathcal{P})$. Next, it identifies the set of *fixed* vertices V_{fixed} in H that are not incident to any removed edge, and determines the subset of pivots among these fixed vertices. The sets R and P are then initialized as $R = V_{\text{fixed}}$ and $P = V(\mathcal{P}) \setminus V_{\text{fixed}}$, respectively. Finally, a recursive Bron-Kerbosch-like search function, `PathUpdate`, is invoked to enumerate all valid subpaths from the residual graph. In each recursive

call, a pivot vertex connected to the largest number of vertices in P is selected [7], and the search expands along its connected candidates to generate the cliques that define the updated path(s).

Correctness. Cases (1)-(2) follow directly from the hold/pivot rules, and case (3) holds because the residual graph deletes exactly the removed pivot–pivot edges while preserving all remaining edges; Bron–Kerbosch enumeration on this residual graph yields all surviving cliques uniquely.

Complexity. Let a clique path $\mathcal{P} = \{V_p(\mathcal{P}), V_h(\mathcal{P})\}$. Let $L = |V_h(\mathcal{P})| + |V_p(\mathcal{P})|$ and let ω_H be the maximum clique size in the residual graph H on $V(\mathcal{P})$. If a hold-hold edge exists, we return in $O(1)$ time. Otherwise the preprocessing on the path is linear, and the only non-trivial part is the local Bron-Kerbosch-like search on H, costing $O(3^{L/3})$ time and $O(L^2)$ space in the worst case, where $L = |V(P)| = |V_h(P)| + |V_p(P)|$

4.3 Case $r = 1$

Similarly as $r = 2$, our algorithm simplifies further for the case of $r = 1$. The update rule for clique paths after removing a vertex is described in Theorem 4.3.

Theorem 4.3 (Vertex Removal)**.** *Let $\mathcal{P}$ be a clique path in the* CPI *of a graph G, with hold and pivot sets denoted by $V_h(\mathcal{P})$ and $V_p(\mathcal{P})$, respectively. For any removed vertex $v \in \mathcal{P}$, the updated $\mathcal{P}'$ satisfies:*

(1) if $v \in V_h(\mathcal{P})$, $\mathcal{P}' = \emptyset$;
(2) if $v \in V_p(\mathcal{P})$, $\mathcal{P}' = \{(V_p(\mathcal{P}) \setminus \{v\},\ V_h(\mathcal{P}))\}$.

Proof. By Lemma 3.1, every k-clique encoded by a path $\mathcal{P}$ must contain all hold vertices on that path, whereas pivot vertices are optional. If the removed vertex v lies in the hold set $V_h(\mathcal{P})$, no encoded k-clique can survive, so the whole path vanishes. If $v \in V_p(\mathcal{P})$, only those cliques that actually include v disappear; the cliques formed by the unchanged hold set and the remaining pivots are still uniquely represented by the shortened path, so we remove v and keep the rest of $\mathcal{P}$ intact. □

Batched Updates. In each iteration of the peeling process, many r-cliques are removed. To avoid redundant computation, we adopt a batched strategy that merges repeated updates. For vertex removals, the algorithm can be easily extended to handle batches. Given a set of vertices to be removed V_{rm} and a path $\mathcal{P}$ containing any of them, we update $\mathcal{P}$ as follows:

(1) $\mathcal{P}' = \emptyset$ if $|V_h(\mathcal{P}) \cap V_{\text{rm}}| > 0$;
(2) $\mathcal{P}' = \{(V_p(\mathcal{P}) \setminus V_{\text{rm}},\ V_h(\mathcal{P}))\}$ if $|V_h(\mathcal{P}) \cap V_{\text{rm}}| = 0$.

We can easily update $\mathcal{P}$ according to the above rules.

Complexity. Let a clique path $\mathcal{P} = \{V_p(\mathcal{P}), V_h(\mathcal{P})\}$. For a single path update, the running time of removing a node from CPI is $O\big(|V_h(\mathcal{P})| + |V_p(\mathcal{P})|\big)$ and the extra space is $O(1)$.

4.4 Complexity.

Table 2 summarizes time/space. Let K_r and K_s be the sets of r- and s-cliques, and let CPI be the clique-path index. The total time consists of (i) build/initial counting, (ii) CPI maintenance during peeling, and (iii) support/heap maintenance.

(i) Build. Building CPI and initializing supports takes $O(|\text{CPI}| + |K_r|)$ (Section 3.3).

(ii) CPI maintenance. Let $T_{\text{UpdateIndex}}$ denote the total cost of maintaining CPI over all peeling steps; it specializes for general r, r=2, and r=1 as analyzed in the previous sections.

(iii) Supports/heap. Let l be the number of peeling rounds (distinct minimum-support values), and let $|P|_r$ be the number of affected r-cliques on path P. Using a bucketed min-heap over K_r, the total time for extracting minima and key updates is $O\big(\big(l + \sum_{P \in \text{CPI}} |P|_r\big) \log |K_r|\big)$.

Worst case. By theorem 3.1, $|\text{CPI}| \le |K_s|$ and $\sum_{P \in \text{CPI}} |P|_r \le s|K_s|$ always holds. But If G is a disjoint union of s-cliques, then $|\text{CPI}| = |K_s|$ and our method degenerates to enumerating all s-cliques.

5 Hierarchy Construction

Thanks to our framework, we can extend the nucleus decomposition to efficiently construct the hierarchy of nuclei. The hierarchy is represented as a tree structure, where each node corresponds to a nucleus and edges denote containment relationships between nuclei. The root of the tree represents the entire graph, while the

Table 2. Complexity summary. κ is arboricity, $ct_r(v) = |\{C_r \in K_r : v \in C_r\}|$, and l is the number of peeling rounds.

Method	Time = *Build/init* + *Support updates* + *Peel-min*
Enum. [44]	$O(m\kappa^{r-2}) + O\big(\sum_{v\in V} ct_r(v)\, deg(v)^{s-r}\big) + O(l \log \lvert K_r\rvert)$
CPI-based	$O(\lvert\mathsf{CPI}\rvert + \lvert K_r\rvert) + \big(T_{\text{UpdateIndex}} + O\big(\sum_{P\in\mathsf{CPI}} \lvert P\rvert_r \log \lvert K_r\rvert\big)\big) + O(l \log \lvert K_r\rvert)$
Space	Enum.: $O(\lvert K_r\rvert)$; CPI: $O(\lvert\mathsf{CPI}\rvert + \lvert K_r\rvert)$

Algorithm 7: UpdateSupport(Hierarchy)

Input: Remove r-clique se R_{min}, minimum heap $H_{\min}$, Clique Path Index $\mathcal{B}$, s-support map *support*
Output: Updated *support* and union-find structure

```
foreach clique path P containing rc ∈ R_min do
    Update support(c) and H_min for all s-clique c ∈ P
    UniteLevel(rc, c, currentK) for all s-clique c ∈ P
Function UniteLevel(a, b, currentK):
    for k ← currentK to 0 do
        if not codeDisjointSets[k].unite(a, b) then
            return;
```

leaves correspond to the smallest, most cohesive nuclei. We formally define hierarchy construction in the context of nucleus decomposition.

Definition 5.1 (s-Connectivity)**.** Given integers $0 < r < s$, two r-cliques R and R' in a graph (or subgraph) H are said to be *s-connected* if there exists a sequence of r-cliques $R = R_1, R_2, \ldots, R_t = R'$, referred to as an *$s$-clique path*, such that for each $i \in \{1, \ldots, t-1\}$ there exists an induced s-clique $S_i \subseteq V(H)$ satisfying $R_i \cup R_{i+1} \subseteq S_i$.

Definition 5.2 (s-Connected Component)**.** For integers $0 < r < s$, the *s-connected component* of an r-clique R is the induced subgraph

$$\text{Comp}_s(R) = G\Big[\bigcup_{R' : R' \text{ is } s\text{-connected to } R} R'\Big].$$

It is the maximal subgraph where all r-cliques are mutually s-connected, and such components partition all r-cliques in G.

Hierarchy can be constructed by identifying containment relationships among nuclei. Specifically, an (r, s)-nucleus N is said to *contain* another (r', s')-nucleus N' if every r'-clique in N' is also an r-clique in N, and $s' \geq s$. This containment relationship is represented as a directed edge from N to N' in the hierarchy tree. The CPI can be used to efficiently determine whether two r-cliques are s-connected.

Theorem 5.1 (Clique Connectivity)**.** *Given Two r-cliques C and C' and a clique path $\mathcal{P}$ of the* CPI*, if $C \subseteq \mathcal{P}$ and $C' \subseteq \mathcal{P}$, then C and C' are s-connected.*

Proof. If $C \subseteq \mathcal{P}$ and $C' \subseteq \mathcal{P}$, then there exists a sequence of r-cliques $C = C^1, C^2, \ldots, C^k = C'$ such that each consecutive pair C^i and C^{i+1} are s-connected. This is because each r-clique in the sequence shares an s-clique with the next one, as they are all part of the same clique path $\mathcal{P}$. Therefore, by the definition of s-connectedness, C and C' are s-connected. □

Based on the above theorem, we can use a multi-level disjoint set union (DSU) data structure to identify all (r, s)-nuclei. Specifically, we maintain a multi-level DSU where each level corresponds to a k-core, and within each level, we maintain the connected components of r-cliques. As we perform the core decomposition, we dynamically update these connected components. The pseudocode is shown in Algorithm 7.

Table 3. Statistics of Datasets.

Dataset	n	m	Avg deg.	Max deg.	#Triangles	Max Clique
com-dblp(DBLP)	317 081	1 049 867	6.6	343	2 224 385	114
web-Stanford(STF)	281 903	1 992 636	14.1	38 625	11 329 473	61
com-youtube(YT)	1 134 890	2 987 624	5.3	28 754	3 056 386	17
web-Google(GO)	875 713	4 322 051	9.9	6 332	13 391 903	44
web-it-2004(WI)	509 338	7 178 413	28.2	469	338 884 212	432
soc-pokec(PO)	1 632 803	22 301 964	27.3	14 854	32 557 458	29

Algorithm. The procedure in Algorithm 7 follows the base pipeline; we keep a stack of DSUs (one per level) to maintain s-components. For each batch $S_{\min}$ assigned level k, `UniteLevel` applies unions on affected CPI paths and replicates each successful union to all levels at most k.
Correctness. A k-(r,s)-nucleus is always a subgraph of a $(k-1)$-(r,s)-nucleus [41]. This implies that if two r-cliques are connected at level k, they must also remain connected at level $k-1$. Therefore, if two r-cliques are found to be connected during peeling by Theorem 5.1, they remain connected at all lower levels. This property defines the behavior of the `UniteLevel` function. For the correctness of the early return (Line 6): if it were incorrect, there would exist a scenario in which two r-cliques are connected in the k-(r,s)-nucleus but not correctly connected in the $(k-1)$-(r,s)-nucleus. Such a situation is impossible because, during the peeling process, $c_{\min}$ is monotonically increasing, and each union operation is consistently propagated to all lower levels.
Complexity. The complexity is identical to that of nucleus decomposition.

6 Experimental Evaluation

Hardware. All algorithms are implemented in C++ and compiled using the g++ compiler with -O3 optimization. Experiments were conducted on a machine equipped with dual Intel(R) Xeon(R) Gold 6342 2.80GHz CPUs, 500GB of RAM, and running Ubuntu 22.
Datasets. We evaluate our approach on six real-world datasets from diverse domains, including Co-authorship Networks: *com-dblp* (cblp) [60]; Social Networks: *soc-pokec* (pokec) [49], *com-youtube* (youtube) [60]; and Web Graphs: *web-Google* (google) [25], *web-it-2004* (web-it) [36], *web-Stanford* (stanford) [25]. These datasets vary in size and structural characteristics, providing a comprehensive evaluation of our approach. Table 3 summarizes the key statistics of these datasets.
Algorithms. We compare our method with two state-of-the-art methods from the ARB line: the peeling framework ARB without hierarchy [44][1] and its hierarchy construction extension [45][2]. and the local h-index framework ND from [40][3]. For ND, in addition to its base implementation, we also evaluate its two optimized parallel variants available in the same repository: the peeling-based PND − peel and the optimized local h-index PND − Opt. ARB, ND and our method support hierarchy construction and core decomposition, while the PND − Opt and PND − peel focus on decomposition. Notably, ARB does not support $r = 1$ due to its algorithmic design. Regarding ND, its base implementation only supports the cases of $(r,s) = (1,3)$, $(1,4)$, $(2,3)$, and $(3,4)$. Furthermore, PND − peel and PND − Opt, are even more specialized and only support the case of $(r,s) = (3,4)$. Our method is publicly available.[4]
Metrics. Each experiment is executed three times, and we report the *average* wall-clock runtime. A timeout of **3600 seconds** (1 hour) is imposed; runs exceeding this limit are marked as *out of time (OOT)* (denoted by ✕ in figures) and excluded from the averages.
Exp-1: Running Time $(r \le 2)$**.** We evaluate $r \in \{1,2\}$ without hierarchy on six datasets, sweeping s from 3/4 up to each dataset's maximum clique size (Figure 5). CND shows little sensitivity to s: curves are essentially flat across datasets. In contrast, ARB and ND grow rapidly with s and often time out. Atypical is `com-youtube`

[1]https://github.com/jeshi96/arb-nucleus-decomp
[2]https://github.com/jeshi96/arb-nucleus-hierarchy
[3]https://github.com/sariyuce/nucleus
[4]https://github.com/JamesZwq/nuclearSigmod/

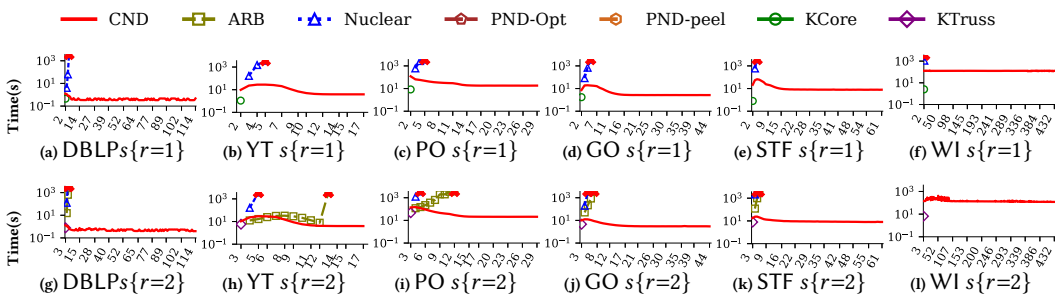


Fig. 5. Running time (seconds) across s without hierarchy for $r = 1$(a-f) and $r = 2$(g-l).

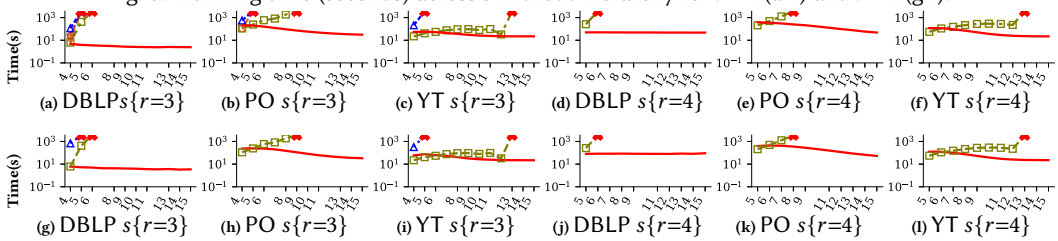


Fig. 6. Running time (seconds) across s for $r \in \{3, 4\}$(a-f without hierarchy; g-l with hierarchy).

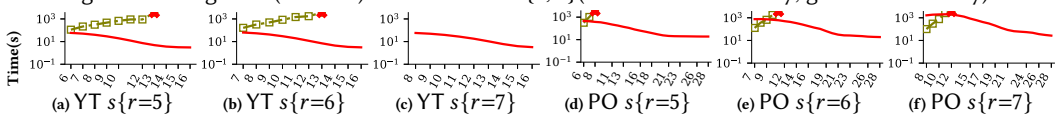


Fig. 7. Running time (seconds) across s without hierarchy for $r \in \{5, 6, 7\}$.

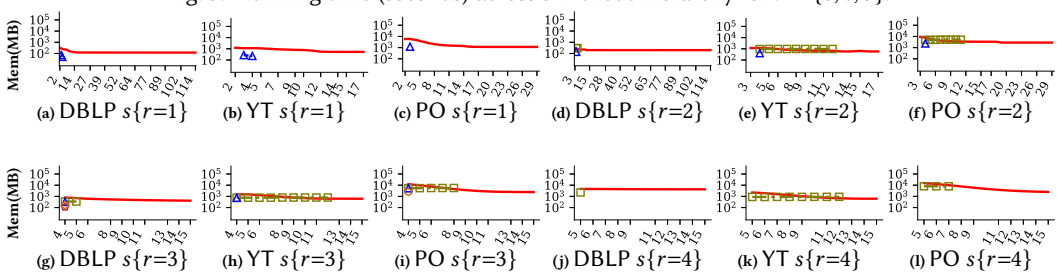
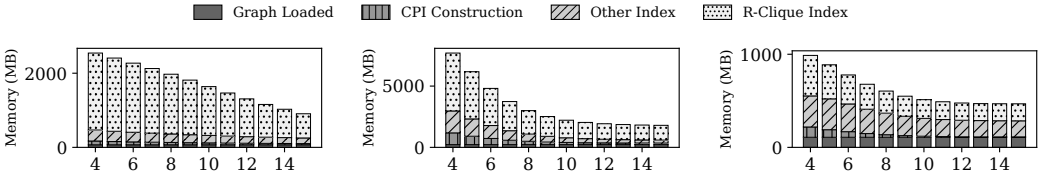


Fig. 8. Time and Memory usage comparison for ($r \in \{1, 2, 3, 4\}$).

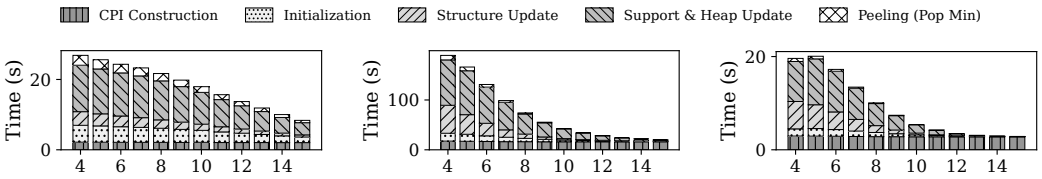
at r=2 (max clique 17), where the gap narrows; even there, ARB times out at s=13 while CND finishes in about 30s. On `web-it-2004` (r=2), ARB times out for all s, whereas CND completes from s=4 onward (e.g., 240s at s=4) and handles the full range. Overall, CND attains average speedups of 30× over ARB and 20× over ND, with a peak 342× on `com-dblp` at r=2, s=5. We further compare against specialized single-thread baselines for the two canonical special cases: k-core ($(r, s) = (1, 2)$)[3] and k-truss ($(r, s) = (2, 3)$)[55]. On `com-dblp`, CND runs in 1.21s for $(1, 2)$ vs. 0.45s for k-core, and 1.31s for $(2, 3)$ vs. 0.78s for k-truss. This indicates that our counting-based framework incurs only modest overhead on the special cases, while preserving the theoretical advantage of avoiding explicit clique enumeration and remaining effective as s increases.

Exp-2: Running Time ($r \in \{3, 4\}$)**.** We evaluate $r \in \{3, 4\}$ on three datasets, with and without hierarchy (Figure 6).



(a) GO $s\{r{=}3\}$ **(b)** PO $s\{r{=}3\}$ **(c)** YT $s\{r{=}3\}$

Fig. 9. Breakdown of **memory usage** on different datasets ($r = 3$) as s varies.



(a) GO $s\{r{=}3\}$ **(b)** PO $s\{r{=}3\}$ **(c)** YT $s\{r{=}3\}$

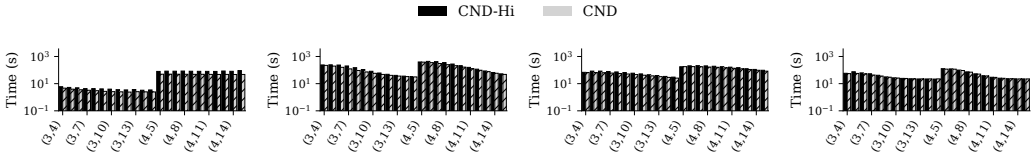
Fig. 10. Breakdown of **running time** on different datasets ($r = 3$) as s varies.

Exp-2.1: Without Hierarchy. Figure 6 compares CND with ND and ARB. As with $r \leq 2$, CND is nearly insensitive to s, curves stay essentially flat, while ARB (and ND) grow rapidly with s and frequently time out. On average, CND is 10× faster than ARB and 9× faster than ND, with a peak 106× speedup on `com-dblp` at r=3, s=5. We also evaluate the specialized algorithms PND − peel and PND − Opt for $(r, s) = (3, 4)$. On `com-dblp`, CND achieves 3.2× and 5.5× speedups, respectively. However, CND is less effective on graphs with small max cliques but large average degrees (e.g., `soc-pokec` and `web-Google`). In these datasets, for $(r, s) = (3, 4)$, PND − Opt, PND − peel, and ARB are all around 1× faster than CND; however, as s increases (e.g., $(3, 5)$ and $(4, 5)$), CND quickly gains a strong advantage. In such cases, the enumeration cost for baselines is naturally low, while the dense connectivity increases the maintenance overhead of our index.

Exp-2.2: With Hierarchy. Figure 6.2 shows the hierarchical setting. The trend persists: CND remains the fastest across all datasets, and its runtime stays flat as s increases, whereas baselines often time out. On average, CND is 6× faster than ARB and 41× faster than ND, with a peak 115× on `com-dblp` at r=3, s=4.

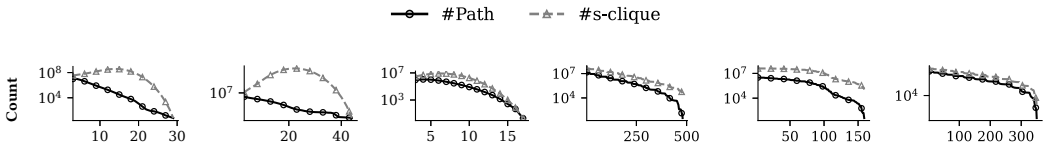
Exp-3: Running Time ($r \in \{5, 6, 7\}$)**.** Figure 7 reports the efficiency for higher-order nucleus decomposition on `soc-pokec` and `com-youtube`. As r increases, CND maintains a robust and relatively flat performance curve. In contrast, ARB exhibits a non-monotonic trend. On `soc-pokec`, the speedup of CND over ARB narrows at medium r (e.g., dropping to 1.0× at r=6), as stricter constraints shrink the search space for enumeration. However, this resilience is transient. At r=7, ARB degrades sharply on `soc-pokec` (e.g., taking 2590s at s=11 vs. 31s for CND, an 83× difference) and eventually times out. The situation is even worse on `com-youtube`, where ARB fails to complete entirely at $r \geq 7$. This demonstrates that while enumeration-based frameworks can transiently benefit from filtered search spaces, they remain fundamentally fragile to high-order complexity.

Exp-4: Memory Usage. Figures 8(a-i) and 9 analyze memory consumption. Our method and the state-of-the-art ARB have comparable memory usage overall. As s increases, our memory decreases while ARB stays nearly constant. This is because larger s leaves fewer valid r-cliques: many r-cliques contain no s-cliques. We do not store such r-cliques, whereas ARB cannot reliably pre-filter them and thus maintains states for all r-cliques. For r=1, our method uses more memory than ND but is substantially faster. On `com-dblp` (r=1, s=4), ND uses 73MB and takes 62.8s, while our method uses 253MB ($\approx$ 3.5×) and finishes in 0.8s (77.7× speedup). On `web-Stanford` (r=1, s=3), ND uses 126MB and runs for 596s, while our method uses 656MB and completes in 55s (x10.8× speedups). Thus, even in low-order settings where baselines are optimized, the additional memory translates to significant performance gains. Figure 9 further breaks down CND's memory usage across datasets and (r, s) settings. Most memory is spent on storing r-cliques rather than CPI, indicating that CPI itself is memory-efficient.



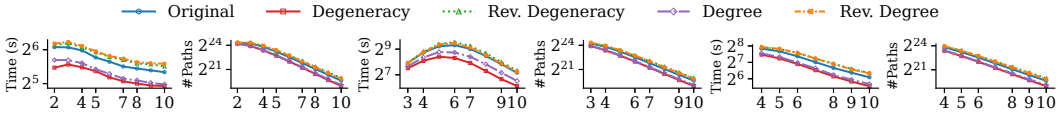
(a) DBLP(r,s) **(b)** PO (r,s) **(c)** GO (r,s) **(d)** YT (r,s)

Fig. 11. Time (in seconds) Hierarchical vs Non-Hierarchical with varying r, s.



(a) PO k-clique **(b)** GO k-clique **(c)** YT k-clique **(d)** PO #Iter **(e)** GO #Iter **(f)** YT #Iter

Fig. 12. CPI #Paths vs. #s-cliques for $r = 3, s = 4$.



(a) $s\{r{=}1\}$ Time **(b)** $s\{r{=}1\}$ #Paths **(c)** $s\{r{=}2\}$ Time **(d)** $s\{r{=}2\}$ #Paths **(e)** $s\{r{=}3\}$ Time **(f)** $s\{r{=}3\}$ #Paths

Fig. 13. Impact of different vertex ordering strategies in soc-pokec

Exp-5: Runtime Breakdown. Figure 10 reports the runtime breakdown of CND across datasets. We decompose the total time into five components: *CPI Construction*, *Init* (initial r-clique enumeration and support counting), *Structure Update* (updating CPI structure during peeling), *Support* (updating s-clique supports via neighbor intersections), and *Peeling* (heap maintenance and PopMin operations). Overall, *Init* and *Support* are the main cost drivers. On `com-dblp` (r=4), *Init* accounts for 36.2%–38.0% and *Support* for 17.7%–18.3% (two runs); on `web-Google` (r=4), *Support* becomes the bottleneck (39.9%) while *Init* is 14.5% (and *Peeling* is 27.3%). These results indicate that runtime is primarily governed by clique enumeration and neighbor intersections (Init/Support), while CPI construction/maintenance and heap-based peeling contribute the remaining overhead.

Exp-6: Ablation Study for $r \in 1, 2$. Figure 14 reports an ablation study for $r \in \{1, 2\}$. To reduce the overhead of generic clique enumeration at low orders, we add specialized optimizations for r=1 and r=2. Compared with the generic baseline (`CBS3`), the optimized version (`CBS-noHi`) is consistently faster: on `com-dblp` with r=2, s=4, runtime drops from 1.57s to 1.04s (1.5×), and for r=1 we observe ≈1.3× speedup. Memory usage remains similar since the optimization only changes CPI updates and does not change the stored CPI structure.

Exp-7: Hierarchy vs. No Hierarchy. Figure 11 compares CND with and without hierarchical construction for $r \in \{3, 4\}$ on `Youtube`, `soc-pokec`, and `web-Google`. Across all three datasets, the two variants have nearly identical runtimes; the hierarchical version is about 10% slower on average. This indicates that our CND algorithm exhibits very stable performance regardless of whether hierarchical construction is employed.

Exp-8: Node Orderings. We evaluate five vertex orderings on `soc-pokec` (Figure 13): `origin`, `degree`/`degreeR`, and `degen`/`degenR`. Across all tested s (2–10 for r=1, 3–10 for r=2, and 4–10 for r=3), `degen` is consistently the best overall. Averaged over all tested s, `degen` is 1.39× faster than `origin` for r=1 (52.6s → 37.9s), 1.86× for r=2 (392.5s → 210.5s), and 1.37× for r=3 (139.8s → 102.1s). In contrast, the reverse orderings are consistently

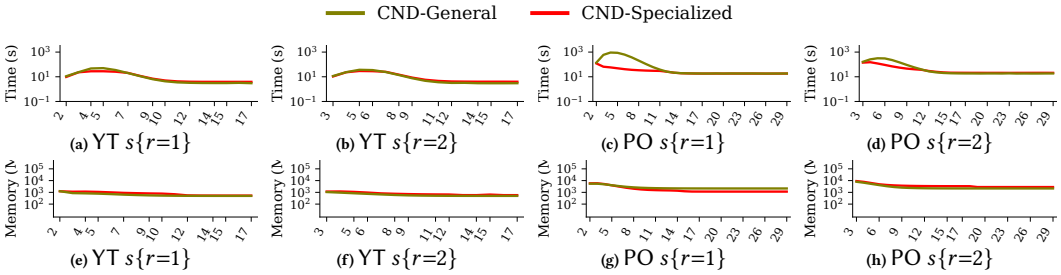


Fig. 14. Comparison of execution time for CND general vs. specialized algorithms.

the slowest: degenR/degreeR are 1.52–2.11× slower than degen on average. Moreover, degen shrinks the CPI: compared with origin, it reduces CPI #paths by 28.3% / 30.9% / 32.5% on average for r=1/2/3 (up to 37.2% at s=10), translating to average memory savings of 0.82 / 0.67 / 0.37 GB. For example, at r=1, s=3, degen reduces CPI #paths from 19.78M to 15.82M (−20%), lowering memory from 6.35 GB to 5.18 GB (−1.17 GB) and runtime from 67.2s to 47.3s. We therefore adopt degen as the default ordering strategy.

Exp-9: #Paths vs. #s-Cliques. Across peeling iterations, we compare the number of CPI paths with the number of s-cliques (Figure 12) for $(r, s) = (3, 4)$ on three datasets. On all datasets and in almost all iterations, the path count is far below $|K_s|$, indicating that CPI stays compact and output-sensitive. Moreover, the path count is flat or decreasing over time, suggesting that path splits are non-inflationary. A mild exception is com-youtube at s=4, whose sparsity and small maximum clique size (ω=17) narrow the gap, consistent with its more modest speedups (Figure 5h).

Exp-10: Stress Test. To evaluate limits under combinatorial explosion, We stress-test on N=1000 synthetic graphs built by incremental clique injection: clique sizes are sampled from a truncated power-law (max size 100), and density is increased from p=0.01, yielding $\bar{d} \approx p(|V| - 1)$. It is worth noting that our starting density p=0.01 is already orders of magnitude higher than typical real-world networks (e.g., com-dblp has $p \approx 2\times10^{-5}$), creating a rigorous worst-case scenario. Experimental results ($r \in \{4, 5\}, s \in \{5..9\}$) demonstrate that CND exhibits superior robustness. As complexity increases, the baseline ARB collapses rapidly: for (r, s)=$(4, 7)$, it times out at a mere p=0.05, and for $s \geq 8$, it fails immediately at p=0.01. In contrast, CND robustly processes densities up to p=0.20 across all settings. For (r, s)=$(4, 6)$ at p=0.11, CND achieves a 210× speedup (30s vs. 6310s). Regarding memory, although CND consumes higher memory than ARB ($\approx$ 2.3×), the usage ratio remains constant as density increases. This linear scaling indicates that our index structure effectively withstands the exponential growth of redundant cliques that paralyzes enumeration-based methods.

Leaf cliques vs. total cliques. As p increases, the peak path in CPI count grows and shifts to larger k (e.g., from 999 at k=3 when p=0.01, to 2.98×10^9 at k=10 when p=0.38), but the total k-clique count grows much faster and becomes infeasible to enumerate. For example, at p=0.01 there are 3.53×10^{10} 20-cliques but only 46 leaf cliques of size 20; at p=0.38 there are 8.41×10^{27} 49-cliques but only 4,028 leaf cliques of size 49. This shows that even under extremely dense graphs, CPI still achieves a high compression ratio.

Suggestion for algorithm selection. Our recommendation is simple: ARB is competitive for *low-order* settings on *very dense* graphs (e.g., r=4, s=5, p>0.18) and uses less memory. In most other regimes—*sparser* graphs (p<0.18) or *higher-order* settings ($r \geq 5, s \geq 6$)—we recommend CND, which consistently achieves large speedups with comparable memory usage to ARB.

Exp-11: Case Study I: Parameter Sensitivity on soc-pokec. We study how r and s control the granularity of (r, s)-nucleus cores on soc-pokec (Figure 16). For each core value k, we report the number of surviving vertices and connected components in the remaining nucleus subgraph, and include k-core (r=1, s=2) and k-truss (r=2, s=3) as baselines. Since the absolute scale of k varies across (r, s), we compare different settings mainly at the same *rank* of k along each curve.

Varying r with fixed s. With s=10, at the 25%-k rank, r=3 keeps 141,392 vertices with 110 components, while r=6 keeps 148,295 vertices but increases to 245 components.

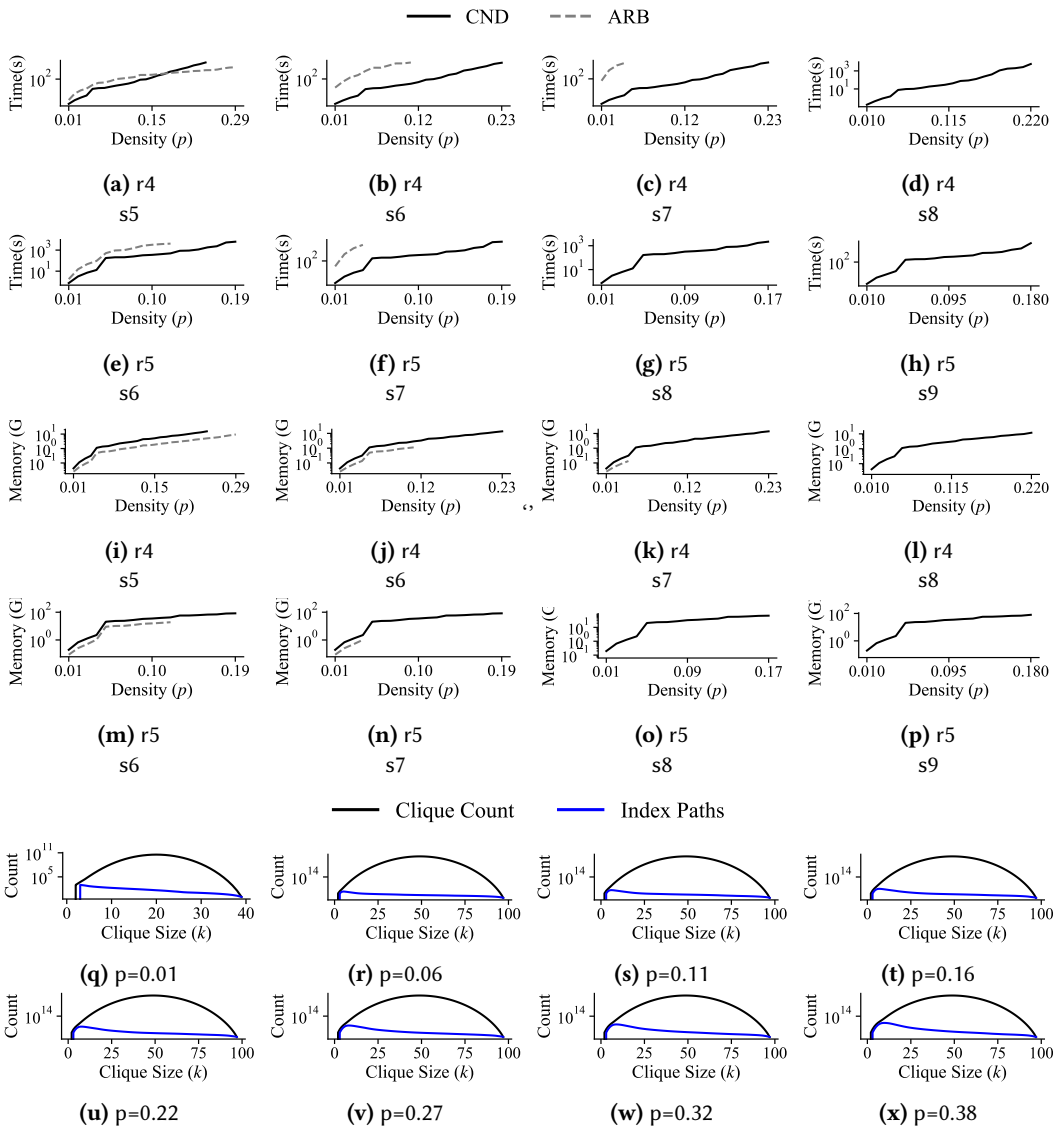


Fig. 15. Comprehensive Stress Test Analysis. **(a-h)** Runtime comparison across various (r, s) pairs; **(i-p)** Memory usage comparison corresponding to the top row; **(q-x)** Leaf clique counts across varying densities (p).

Varying s with fixed r. Fix r=3 and increase s from 10 to 14 to 18. At the 10%-k rank (a typical operating regime), the remaining subgraph shrinks from 195,278 vertices / 226 components (s=10) to 32,310 / 30 (s=14) and further to 5,009 / 8 (s=18). At the median k rank, vertices drop from 78,233 / 41 (s=10) to 11,325 / 14 (s=14) and 2,867 / 5 (s=18).

k-core and k-truss (low-order baselines). At the 10%-k rank, k-core retains 18,449,233 vertices in a single component, which is too coarse to separate modules. k-truss retains 10,138,083 vertices but yields 14,286 components, indicating that low-order $(2, 3)$ support admits many residual structures. By contrast, $(r, s) = (3, 10)$ at the same

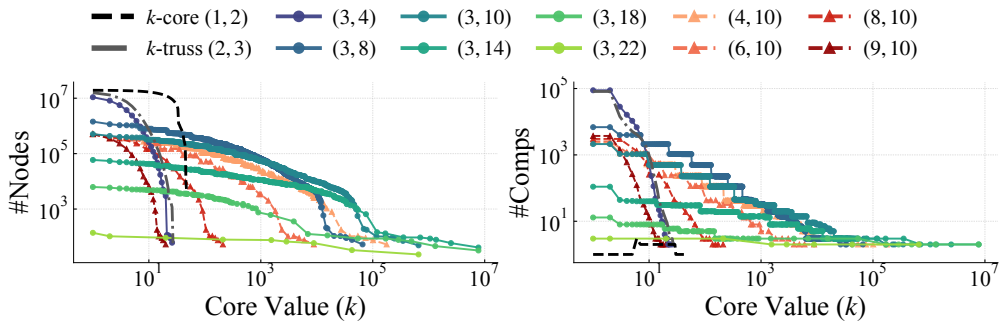


Fig. 16. Case study on soc-pokec. Solid (o): varying s; dashed (△): varying r.

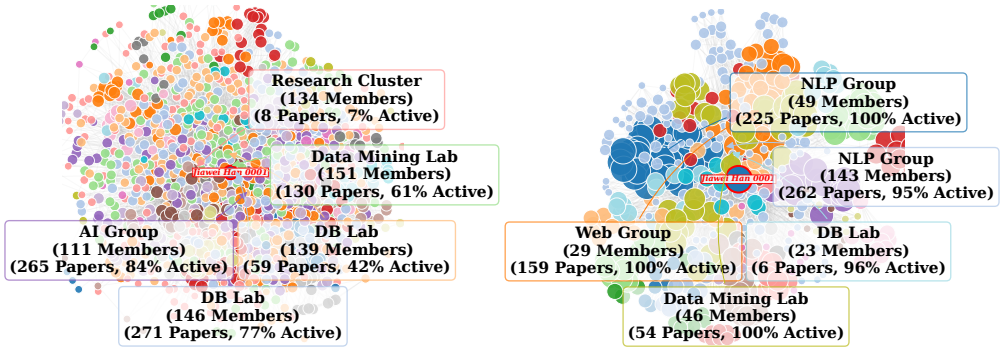


Fig. 17. Case Study on Collaborative network: (a) Low Quality, (b) High Purity.

rank retains 195,278 vertices and 226 components, producing substantially more compact and well-structured modules.

The benefit of tuning s. Increasing s reduces both the surviving size and the number of components, producing compact and well structured modules. It theoretically filters out incidental structures.

Practical guideline. We recommend fixing a moderate r (e.g., r=3) and tuning s to control granularity. When s is too large(s=22), close to the dataset's maximum clique size (29), very few s-cliques exist and the remaining subgraph may become too small to be informative, most nodes are removed, leaving only a few nodes and components.

Exp-12: Case Study II: Identifying ego network on DBLP.. We study Jiawei Han's 1-hop coauthor ego graph on DBLP and compare s=4 vs. s=10 under r=3. In Fig. 17, nodes are colored by nucleus component and we annotate the top-5 largest ones. For each surviving component C, let m_{in} be the number of internal edges and m_{cut} the number of edges from C to other surviving components. We report $\phi(C) = m_{\text{cut}}/(2m_{\text{in}} + m_{\text{cut}})$ and $m_{\text{in}}/m_{\text{cut}}$; lower $\phi(C)$ and higher $m_{\text{in}}/m_{\text{cut}}$ indicate better separation. Using DBLP paper lists restricted to this ego network, a paper is *collaborative* for C if it contains ≥ 3 authors from C, and the *active ratio* is the fraction of members appearing in at least one collaborative paper.

Larger s yields compact and better-separated modules. Increasing s from 4 to 10 shrinks the surviving subgraph from 1,328 to 540 nodes and reduces components from 65 to 27, while increasing the intra-component edge fraction from 0.319 to 0.528. The size-weighted average separability improves from 0.270 to 0.750, and the conductance score drops from 0.712 to 0.475, showing that larger s produces more cleanly separated collaboration modules.

Larger s preserves highly active. Under the same collaboration threshold (3 within-component coauthors), the top-5 annotated components at s=4 have an average active ratio of 54.25%, and already exhibit a clear long tail (e.g., a 134-member component has only 8 collaborative papers and 7% active members). In contrast, at s=10

the top-8 components have an average active ratio of 98.15%; most are near 100% active, with the smallest 95%. This indicates that increasing s filters loosely coordinated/noisy groups and retains compact, strongly collaborating teams that are easier to interpret.

7 Related Work

Cohesive subgraph discovery is fundamental in graph mining. Classic peeling-based models include k-core decomposition [11, 21, 38, 43, 57, 59, 63–65], which removes vertices with degree $< k$, and k-truss decomposition [1, 8, 18, 50, 55, 56, 58, 66], which removes edges whose triangle support is $< k$. Nucleus decomposition was introduced by [41], generalizing k-core$(1, 2)$ and k-truss$(2, 3)$ to (r, s)-nuclei; other nucleus-decomposition works are discussed earlier and not repeated. Dense-subgraph mining and quasi-clique discovery target objective-driven dense regions or relaxed clique constraints [22, 24]; they are complementary to (r, s)-nuclei, which are defined via higher-order clique containment/support and induce a global hierarchy across core values. Community search is query-dependent, returning a cohesive subgraph containing given query vertices [46, 48]; it emphasizes efficient per-query answers (often under constraints such as size/connectivity), whereas we focus on global (r, s)-nucleus peeling with efficient dynamic index updates. Finally, clique enumeration/counting are key primitives for higher-order cohesiveness; recent work revisits scalable k-clique counting by combining exact counting with sampling [10, 23, 61, 62].

8 Conclusion and future work

In this paper, we revisited nucleus decomposition and presented the first *counting-based* framework that eliminates explicit s-clique enumeration. Our key structure is the *Clique Path Index* (CPI), which organizes the s-clique search space to enable direct counting, efficient dynamic updates, and connectivity maintenance during decomposition. Extensive experiments show substantial speedups for both decomposition and hierarchy construction, and strong scalability to larger s.

Our counting-and-indexing viewpoint suggests natural future directions. First, it is amenable to parallelization: BuildIndex / InitialCounting parallelize over pivot vertices (or seed r-cliques) with a final reduction, while UpdateIndex/UpdateSupport can batch removed r-cliques using thread-local buffers and aggregated (atomic) support updates. More broadly, CPI abstracts to indexing containment between small patterns and their supporting large patterns so that supports are maintained by counting, not explicit enumeration. This extends beyond cliques [13] to bicliques in bipartite graphs (related to (α, β)-core) [51], hyperedge-induced group patterns in hypergraphs [28, 29, 64], and temporal Δ-cliques [9, 54]. We leave optimized multi-threaded implementations and these extensions as future work.

9 Acknowledgements

Dong Wen is supported by ARC DP230101445 and ARC DE240100668. Wenjie Zhang is supported by ARC DP230101445 and ARC FT210100303. Xuemin Lin is supported by NSFC U2241211, NSFC U20B2046, and GuangDong Basic and Applied Basic Research Foundation 2019B1515120048.

References

[1] Esra Akbas and Peixiang Zhao. 2017. Truss-based community search: a truss-equivalence based indexing approach. *Proc. VLDB Endow.* 10, 11 (Aug. 2017), 1298–1309. doi:10.14778/3137628.3137640

[2] Leman Akoglu, Hanghang Tong, and Danai Koutra. 2015. Graph-based anomaly detection and description: a survey. *Data Mining and Knowledge Discovery* 29, 3 (2015), 626–688.

[3] V. Batagelj and M. Zaversnik. 2003. An O(m) Algorithm for Cores Decomposition of Networks. arXiv:cs/0310049 [cs.DS] https://arxiv.org/abs/cs/0310049

[4] Vladimir Batagelj and Matjaž Zaveršnik. 2011. Fast algorithms for determining (generalized) core groups in social networks. *Adv. Data Anal. Classif.* 5, 2 (July 2011), 129–145. doi:10.1007/s11634-010-0079-y

[5] Austin R. Benson, David F. Gleich, and Jure Leskovec. 2016. Higher-order organization of complex networks. *Science* 353, 6295 (July 2016), 163–166. doi:10.1126/science.aad9029

[6] Alex Beutel, Wanhong Xu, Venkatesan Guruswami, Christopher Palow, and Christos Faloutsos. 2013. CopyCatch: stopping group attacks by spotting lockstep behavior in social networks. In *Proceedings of the 22nd International Conference on World Wide Web* (Rio de Janeiro, Brazil) *(WWW '13)*. Association for Computing Machinery, New York, NY, USA, 119–130. doi:10.1145/2488388.2488400

[7] Coen Bron and Joep Kerbosch. 1973. Algorithm 457: finding all cliques of an undirected graph. *Commun. ACM* 16, 9 (Sept. 1973), 575–577. doi:10.1145/362342.362367

[8] Yulin Che, Zhuohang Lai, Shixuan Sun, Yue Wang, and Qiong Luo. 2020. Accelerating truss decomposition on heterogeneous processors. *Proc. VLDB Endow.* 13, 10 (June 2020), 1751–1764. doi:10.14778/3401960.3401971

[9] Kaiyu Chen, Dong Wen, Wentao Li, Zhengyi Yang, and Wenjie Zhang. 2024. On compressing historical cliques in temporal graphs. In *International Conference on Database Systems for Advanced Applications*. Springer Nature Singapore Singapore, 37–53.

[10] Kaiyu Chen, Dong Wen, Hanchen Wang, Zhengyi Yang, Wenjie Zhang, and Xuemin Lin. 2025. Covering K-Cliques in Billion-Scale Graphs. In *The Web Conference 2025*.

[11] James Cheng, Yiping Ke, Shumo Chu, and M. Tamer Özsu. 2011. Efficient core decomposition in massive networks. In *2011 IEEE 27th International Conference on Data Engineering*. International Conference on Data Engineering, Hannover, 51–62. doi:10.1109/ICDE.2011.5767911

[12] Shumo Chu and James Cheng. 2012. Triangle listing in massive networks. *ACM Trans. Knowl. Discov. Data* 6, 4, Article 17 (Dec. 2012), 32 pages. doi:10.1145/2382577.2382581

[13] Reinhard Diestel. 2000. *Graph Theory* (2 ed.). Springer.

[14] David Gibson, Jon Kleinberg, and Prabhakar Raghavan. 1998. Inferring Web communities from link topology. In *Proceedings of the Ninth ACM Conference on Hypertext and Hypermedia: Links, Objects, Time and Space—Structure in Hypermedia Systems: Links, Objects, Time and Space—Structure in Hypermedia Systems* (Pittsburgh, Pennsylvania, USA) *(HYPERTEXT '98)*. Association for Computing Machinery, New York, NY, USA, 225–234. doi:10.1145/276627.276652

[15] M. Girvan and M. E. J. Newman. 2002. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences* 99, 12 (2002), 7821–7826. arXiv:https://www.pnas.org/doi/pdf/10.1073/pnas.122653799 doi:10.1073/pnas.122653799

[16] Wolfgang Glänzel and András Schubert. 2005. *Analysing Scientific Networks Through Co-Authorship*. Springer Netherlands, Dordrecht, 257–276. doi:10.1007/1-4020-2755-9_12

[17] Bryan Hooi, Hyun Ah Song, Alex Beutel, Neil Shah, Kijung Shin, and Christos Faloutsos. 2016. FRAUDAR: Bounding Graph Fraud in the Face of Camouflage. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. Association for Computing Machinery, New York, NY, USA, 895–904. doi:10.1145/2939672.2939747

[18] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying k-truss community in large and dynamic graphs. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 1311–1322. doi:10.1145/2588555.2610495

[19] Shweta Jain and C. Seshadhri. 2020. The Power of Pivoting for Exact Clique Counting. In *Proceedings of the 13th International Conference on Web Search and Data Mining* (Houston, TX, USA) *(WSDM '20)*. Association for Computing Machinery, New York, NY, USA, 268–276. doi:10.1145/3336191.3371839

[20] David Kempe, Jon Kleinberg, and Éva Tardos. 2003. Maximizing the spread of influence through a social network. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Washington, D.C.) *(KDD '03)*. Association for Computing Machinery, New York, NY, USA, 137–146. doi:10.1145/956750.956769

[21] Wissam Khaouid, Marina Barsky, Venkatesh Srinivasan, and Alex Thomo. 2015. K-core decomposition of large networks on a single PC. *Proc. VLDB Endow.* 9, 1 (2015), 13–23. doi:10.14778/2850469.2850471

[22] Aritra Konar and Nicholas D. Sidiropoulos. 2024. Optimal quasi-clique: hardness, equivalence with densest-k-subgraph, and quasi-partitioned community mining. In *Proceedings of the Thirty-Eighth AAAI Conference on Artificial Intelligence*

and Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence and Fourteenth Symposium on Educational Advances in Artificial Intelligence (AAAI'24/IAAI'24/EAAI'24). AAAI Press, Article 957, 9 pages. doi:10.1609/aaai.v38i8.28705

[23] Longbin Lai, Zhu Qing, Zhengyi Yang, Xin Jin, Zhengmin Lai, Ran Wang, Kongzhang Hao, Xuemin Lin, Lu Qin, Wenjie Zhang, et al. 2019. Distributed subgraph matching on timely dataflow. *Proceedings of the VLDB Endowment* (2019).

[24] Tommaso Lanciano, Atsushi Miyauchi, Adriano Fazzone, and Francesco Bonchi. 2024. A Survey on the Densest Subgraph Problem and its Variants. *ACM Comput. Surv.* 56, 8, Article 208 (April 2024), 40 pages. doi:10.1145/3653298

[25] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. 2008. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. arXiv:0810.1355 [cs.DS] https://arxiv.org/abs/0810.1355

[26] Jure Leskovec and Rok Sosič. 2016. SNAP: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology* 8, 1 (2016), 1.

[27] Greg Linden, Brent Smith, and Jeremy York. 2003. Amazon.com Recommendations: Item-to-Item Collaborative Filtering. *IEEE Internet Computing* 7, 1 (Jan. 2003), 76–80. doi:10.1109/MIC.2003.1167344

[28] Qi Luo, Wenjie Zhang, Zhengyi Yang, Dong Wen, Xiaoyang Wang, Dongxiao Yu, and Xuemin Lin. 2024. Hierarchical structure construction on hypergraphs. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management*. 1597–1606.

[29] Qi Luo, Wenjie Zhang, Zhengyi Yang, Dongxiao Yu, Xuemin Lin, and Liping Wang. 2025. Efficient indexing and searching of constrained core in hypergraphs. *The VLDB Journal* 34, 3 (2025), 34.

[30] Fragkiskos D. Malliaros, Christos Giatsidis, Dimitrios M. Thilikos, and Michalis Vazirgiannis. 2020. The core decomposition of networks: theory, algorithms and applications. *The VLDB Journal* 29, 1 (2020), 61–92. doi:10.1007/s00778-019-00587-4

[31] Fragkiskos D. Malliaros and Michalis Vazirgiannis. 2013. Clustering and community detection in directed networks: A survey. *Physics Reports* 533, 4 (2013), 95–142. doi:10.1016/j.physrep.2013.08.002 Clustering and Community Detection in Directed Networks: A Survey.

[32] Md Moniruzzaman Monir and Ahmet Erdem Sarıyüce. 2020. Using Large Cliques for Hierarchical Dense Subgraph Discovery. In *International Conference on Computational Data and Social Networks*. Springer, 179–192.

[33] M. E. J. Newman. 2001. The structure of scientific collaboration networks. *Proceedings of the National Academy of Sciences* 98, 2 (2001), 404–409. arXiv:https://www.pnas.org/doi/pdf/10.1073/pnas.98.2.404 doi:10.1073/pnas.98.2.404

[34] Francis J O'Reilly, Andrea Graziadei, Christian Forbrig, Rica Bremenkamp, Kristine Charles, Swantje Lenz, Christoph Elfmann, Lutz Fischer, Jörg Stülke, and Juri Rappsilber. 2023. Protein complexes in cells by AI‐assisted structural proteomics. *Molecular Systems Biology* 19, 4 (2023), e11544. arXiv:https://www.embopress.org/doi/pdf/10.15252/msb.202311544 doi:10.15252/msb.202311544

[35] Giulio Rossetti and Rémy Cazabet. 2018. Community Discovery in Dynamic Networks: A Survey. *ACM Comput. Surv.* 51, 2, Article 35 (Feb. 2018), 37 pages. doi:10.1145/3172867

[36] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. https://networkrepository.com

[37] Ryan A. Rossi, Nesreen K. Ahmed, Rong Zhou, and Hoda Eldardiry. 2018. Interactive Visual Graph Mining and Learning, Vol. 9. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3200764

[38] Ahmet Erdem Saríyüce, Buğra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V. Çatalyürek. 2013. Streaming algorithms for k-core decomposition. *Proc. VLDB Endow.* 6, 6 (2013), 433–444. doi:10.14778/2536336.2536344

[39] Ahmet Erdem Sariyüce and Ali Pinar. 2016. Fast hierarchy construction for dense subgraphs. *Proc. VLDB Endow.* 10, 3 (Nov. 2016), 97–108. doi:10.14778/3021924.3021927

[40] Ahmet Erdem Sariyüce, C. Seshadhri, and Ali Pinar. 2018. Local algorithms for hierarchical dense subgraph discovery. *Proc. VLDB Endow.* 12, 1 (Sept. 2018), 43–56. doi:10.14778/3275536.3275540

[41] Ahmet Erdem Sariyuce, C. Seshadhri, Ali Pinar, and Umit V. Catalyurek. 2015. Finding the Hierarchy of Dense Subgraphs using Nucleus Decompositions. In *Proceedings of the 24th International Conference on World Wide Web* (Florence, Italy) *(WWW '15)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 927–937. doi:10.1145/2736277.2741640

[42] Ahmet Erdem Sariyüce, C. Seshadhri, Ali Pinar, and Ümit V. Çatalyürek. 2017. Nucleus Decompositions for Identifying Hierarchy of Dense Subgraphs. *ACM Trans. Web* 11, 3, Article 16 (July 2017), 27 pages. doi:10.1145/3057742

[43] Stephen B. Seidman. 1983. Network structure and minimum degree. *Social Networks* 5, 3 (1983), 269–287. doi:10.1016/0378-8733(83)90028-X

[44] Jessica Shi, Laxman Dhulipala, and Julian Shun. 2021. Theoretically and practically efficient parallel nucleus decomposition. *Proc. VLDB Endow.* 15, 3 (Nov. 2021), 583–596. doi:10.14778/3494124.3494140

[45] Jessica Shi, Laxman Dhulipala, and Julian Shun. 2024. Parallel Algorithms for Hierarchical Nucleus Decomposition. *Proc. ACM Manag. Data* 2, 1, Article 32 (March 2024), 27 pages. doi:10.1145/3639287

[46] Mauro Sozio and Aristides Gionis. 2010. The Community-search Problem and How to Plan a Successful Cocktail Party. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. doi:10.1145/1835804.1835923

[47] K. Subramanyam. 1983. Bibliometric studies of research collaboration: A review. *Journal of Information Science* 6, 1 (1983), 33–38. arXiv:https://doi.org/10.1177/016555158300600105 doi:10.1177/016555158300600105

[48] Longxu Sun, Xin Huang, Jiannan Wang, and Jianliang Xu. 2025. A Flexible Framework for Query-Oriented Interactive Community Search. *Proc. VLDB Endow.* 18, 6 (Feb. 2025), 1977–1990. doi:10.14778/3725688.3725720

[49] Lubos Takac. 2012. DATA ANALYSIS IN PUBLIC SOCIAL NETWORKS. https://api.semanticscholar.org/CorpusID:18659578

[50] Anxin Tian, Alexander Zhou, Yue Wang, and Lei Chen. 2023. Maximal D-Truss Search in Dynamic Directed Graphs. *Proc. VLDB Endow.* 16, 9 (May 2023), 2199–2211. doi:10.14778/3598581.3598592

[51] Anxin Tian, Alexander Zhou, Yue Wang, Xun Jian, and Lei Chen. 2024. Efficient Index for Temporal Core Queries over Bipartite Graphs. *Proc. VLDB Endow.* 17, 11 (July 2024), 2813–2825. doi:10.14778/3681954.3681965

[52] 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 (Oct. 2006), 28–42. doi:10.1016/j.tcs.2006.06.015

[53] Johan Ugander, Lars Backstrom, Cameron Marlow, and Jon Kleinberg. 2012. Structural diversity in social contagion. *Proceedings of the National Academy of Sciences* 109, 16 (2012), 5962–5966. arXiv:https://www.pnas.org/doi/pdf/10.1073/pnas.1116502109 doi:10.1073/pnas.1116502109

[54] Tiphaine Viard, Matthieu Latapy, and Clémence Magnien. 2016. Computing maximal cliques in link streams. *Theor. Comput. Sci.* 609, P1 (Jan. 2016), 245–252. doi:10.1016/j.tcs.2015.09.030

[55] Jia Wang and James Cheng. 2012. Truss decomposition in massive networks. *Proc. VLDB Endow.* 5, 9 (May 2012), 812–823. doi:10.14778/2311906.2311909

[56] Yue Wang, Ruiqi Xu, Xun Jian, Alexander Zhou, and Lei Chen. 2022. Towards distributed bitruss decomposition on bipartite graphs. *Proc. VLDB Endow.* 15, 9 (May 2022), 1889–1901. doi:10.14778/3538598.3538610

[57] Dong Wen, Lu Qin, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2016. I/O efficient core graph decomposition at web scale. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 133–144.

[58] Tianyang Xu, Zhao Lu, and Yuanyuan Zhu. 2022. Efficient Triangle-Connected Truss Community Search in Dynamic Graphs. *Proc. VLDB Endow.* 16, 3 (Nov. 2022), 519–531. doi:10.14778/3570690.3570701

[59] Bohua Yang, Dong Wen, Lu Qin, Ying Zhang, Lijun Chang, and Rong-Hua Li. 2019. Index-Based Optimal Algorithm for Computing K-Cores in Large Uncertain Graphs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 64–75. doi:10.1109/ICDE.2019.00015

[60] Jaewon Yang and Jure Leskovec. 2012. Defining and Evaluating Network Communities based on Ground-truth. arXiv:1205.6233 [cs.SI] https://arxiv.org/abs/1205.6233

[61] Zhengyi Yang, Longbin Lai, Xuemin Lin, Kongzhang Hao, and Wenjie Zhang. 2021. Huge: An efficient and scalable subgraph enumeration system. In *Proceedings of the 2021 international conference on management of data*. 2049–2062.

[62] Xiaowei Ye, Rong-Hua Li, Qiangqiang Dai, Hongzhi Chen, and Guoren Wang. 2022. Lightning Fast and Space Efficient k-clique Counting. In *Proceedings of the ACM Web Conference 2022* (Virtual Event, Lyon, France) *(WWW '22)*. Association for Computing Machinery, New York, NY, USA, 1191–1202. doi:10.1145/3485447.3512167

[63] Michael Yu, Dong Wen, Lu Qin, Ying Zhang, Wenjie Zhang, and Xuemin Lin. 2021. On Querying Historical K-Cores. *Proc. VLDB Endow.* 14, 11 (2021), 2033–2045. doi:10.14778/3476249.3476260

[64] Wenqian Zhang, Zhengyi Yang, Dong Wen, Wentao Li, Wenjie Zhang, and Xuemin Lin. 2025. Accelerating Core Decomposition in Billion-Scale Hypergraphs. *Proc. ACM Manag. Data* 3, 1, Article 6 (Feb. 2025), 27 pages. doi:10.1145/3709656

[65] Wenqian Zhang, Zhengyi Yang, Dong Wen, and Xiaoyang Wang. 2023. Efficient Distributed Core Graph Decomposition. In *2023 IEEE International Conference on Data Mining Workshops (ICDMW)*. IEEE, 1023–1031.

[66] Feng Zhao and Anthony K. H. Tung. 2012. Large scale cohesive subgraphs discovery for social network visual analysis. *Proc. VLDB Endow.* 6, 2 (Dec. 2012), 85–96. doi:10.14778/2535568.2448942

[67] Zhengzhang Zhu, Chen Lin, Xuejian Song, Jiuxin Cao, Wei Wang, and Albert Y. Zomaya. 2021. Efficient truss decomposition for large-scale graphs. *IEEE Transactions on Knowledge and Data Engineering* 33, 10 (2021), 3343–3356. doi:10.1109/TKDE.2020.2978311

Received October 2025; revised January 2026; accepted February 2026