Efficient Locality-based Indexing for Cohesive Subgraphs Discovery in Hypergraphs

Song Kim
song.kim@unist.ac.kr
UNIST
Ulsan, South Korea

Dahee Kim
dahee@unist.ac.kr
UNIST
Ulsan, South Korea

Taejoon Han
cheld7132@unist.ac.kr
UNIST
Ulsan, South Korea

Junghoon Kim*
junghoon.kim@unist.ac.kr
UNIST
Ulsan, South Korea

Hyun Ji Jeong
hjjeong@kongju.ac.kr
Kongju National University
Cheonan, South Korea

Jungeun Kim
jekim@inha.ac.kr
Inha University
Incheon, South Korea

Abstract

Hypergraphs, increasingly utilised to model diverse relationships in modern networks, have gained significant attention for representing intricate higher-order interactions. Among various challenges, cohesive subgraph discovery is one of the fundamental problems and offers deep insights into these structures, yet the task of selecting appropriate parameters is an open question. To address this question, we aim to design an efficient indexing structure to retrieve cohesive subgraphs in an online manner. The main idea is to enable the discovery of corresponding structures within a reasonable time without the need for exhaustive graph traversals. Our method enables faster and more effective retrieval of cohesive structures, which supports decision-making in applications that require online analysis of large-scale hypergraphs. Through extensive experiments on real-world networks, we demonstrate the superiority of our proposed indexing technique.

Keywords

Cohesive subgraph discovery, Hypergraph mining, Indexing tree

1 INTRODUCTION

Modelling relationships among multiple entities intuitively and effectively is a fundamental challenge in network analysis. Hypergraphs offer a powerful solution by capturing higher-order relationships among groups of entities, overcoming the pairwise restriction of traditional graphs. Many real-world networks naturally fit into the hypergraph framework, such as co-authorship networks [28, 35], co-purchase networks [39, 42], and location-based social networks [27].

Since hypergraphs capture higher-order interactions, hypergraph mining has become an active area of research. In particular, cohesive subhypergraph discovery has been actively researched, with various models being proposed. These include the k-hypercore [29], (k, l)-hypercore [30], (k, t)-hypercore [9], nbr-k-core [3], (k, d)-core [3], and (k, g)-core [22]. The most recent model, the (k, g)-core, defines a maximal subhypergraph where each node has at least k neighbours appearing together in at least g hyperedges. Unlike previous models, which primarily focus on hyperedge cardinality, the (k, g)-core captures direct cohesiveness among neighbouring nodes. Building on these properties, the model offers deeper structural insight by enabling fine-grained distinctions in how nodes participate across various numbers of hyperedges. Accordingly, it reveals subhypergraphs with more expressive and cohesive structure.

Given the benefits, the (k, g)-core has broad applicability in various domains: (1) Team formation: In scenarios of collaborative work or developing marketing strategies, the (k, g)-core can identify appropriate groups of participants that are closely connected through multiple shared projects or interactions, helping to form cohesive and effective teams. (2) Community Search: Query-centric community discovery is a central task in social network analysis [37]. It mainly relies on models specifically designed for that purpose. Cohesive subgraph models like the (k, g)-core offer a robust framework for identifying tightly-connected substructures within a network [2, 15, 23, 32, 41]. (3) Foundation for other hypergraph mining tasks: The (k, g)-core can support tasks such as identifying densest subhypergraphs [20], influence maximisation [3], and centrality measures [10]. Its robust structure enables optimising information spread and quantifying node importance in complex networks.

Despite its usefulness, selecting appropriate parameters for the cohesive subgraphs remains a significant challenge [12], particularly in the absence of prior knowledge about the network. The difficulty is exacerbated in models such as the (k, g)-core, which incorporate two independent parameters to characterise node-level cohesion. As a result, users are often required to issue repeated and tedious queries, which limits practicality of the model. This issue is especially problematic since many real-world applications require online query processing [18], where the ability to rapidly adjust parameters based on user-specified criteria is critical.

To address this challenge, many cohesive subgraph models employ a technique known as decomposition [6, 17, 38]. This approach stores the network in terms of its core structure, enabling users to obtain query results immediately without redundant computation. While decomposition has become a widely used strategy for cohesive subgraph models, we propose a novel index-based (k, g)-core decomposition method, extending prior approaches with several key advantages. First, it enables efficient query processing by eliminating the need for repeated subgraph computations. Second, it constructs a memory-efficient data structure to represent decomposition results. We will show that this is particularly beneficial for models like the (k, g)-core that exhibit strong locality. Third, the proposed index structures capture latent structural properties of node participation, thereby enhancing interpretability of hypergraph topology. These advantages make the index-based decomposition framework well-suited for a range of applications, including the following: (1) Fraud Detection: In domains such as e-commerce or financial networks, fraudsters

*Corresponding author.

EDBT '26, Tampere (Finland)

10.48786/edbt.2026.15

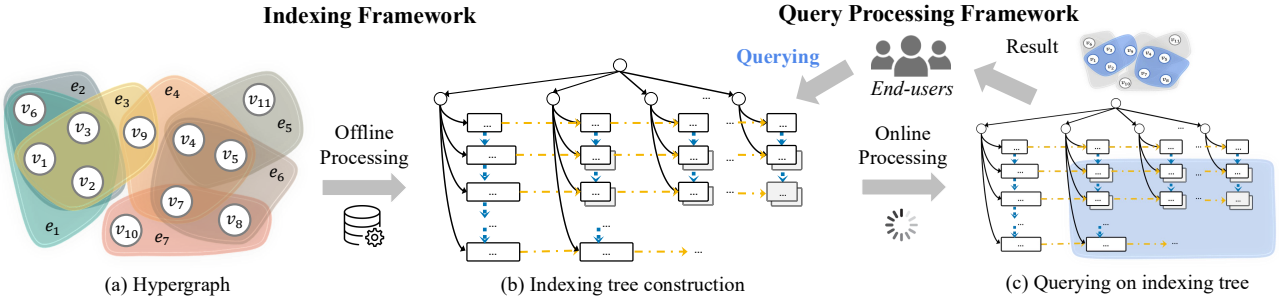


Figure 1: An illustrative example of proposed framework

often operate within tightly knit groups that exhibit suspiciously high levels of interaction [19, 36]. The (k, g)-core can be applied to identify these densely connected groups of fraudulent actors, based on the frequency and nature of their interactions. By leveraging the indexing-based structure, it becomes feasible to detect such activities in online time, enhancing the ability to prevent fraud before it escalates. (2) User Engagement Analysis: Social networks are often analysed to understand user engagement at multiple levels. By using the (k, g)-core and its index structure, we can evaluate user interactions across various dimensions, helping to identify which users are central to the network cohesion at different levels of connectivity. This insight can be useful for recognising influential users, understanding engagement trends, or even detecting abnormal behaviours like bot activity or coordinated misinformation campaigns. (3) Size-Bounded Cohesive Subgraph Discovery: In scenarios like size-bounded team formation, conference planning, or social event organisation, it is crucial to assemble groups of participants who are closely connected and of appropriate community size [32, 41]. The index-based approach can efficiently identify candidate groups that meet these criteria.

In this paper, we propose index-based (k, g)-core decomposition techniques, along with efficient online query processing algorithms that leverage these indexing structures. The overall framework of our proposed approach is illustrated in Figure 1. It has two main phases: (1) Indexing : the hypergraph is preprocessed into a memory-efficient index structure; and (2) Query Processing : users can perform online queries by dynamically adjusting parameters based on the precomputed index.

Challenges. Constructing an efficient index for the (k, g)-core in hypergraphs raises challenges that differ from graph-based core indexing. In particular, the following issues must be addressed:

- **Indexing under dual parameters:** Unlike traditional graph models such as the k-core or k-truss, hypergraphs impose simultaneous constraints on both node degree and group size. No established indexing methods exist for such multi-parameter settings. A straightforward approach based on clique reduction [21] leads to substantial size inflation, and high-cardinality hyperedges exacerbate memory consumption and complicate index maintenance.
- **Space efficiency:** The index must support queries for all (k, g) combinations. Due to the hierarchical nature of the (k, g)-core, cores with nearby parameter values often overlap substantially, resulting in redundancy. A compact design that minimises duplication while preserving query capability is required.
- **Scalable query processing:** The peeling algorithm for (k, g)-core discovery is prohibitively slow on large hypergraphs. The index must thus guarantee query scalability, ensuring practical query latency even at large scale.

Contributions. Index-based cohesive subgraph discovery has been extensively studied in diverse graph settings [15, 31, 33, 43]. Our work leverages hierarchical compression from prior indices while addressing several key distinctions. Specifically, the (k, g)-core on hypergraphs introduces simultaneous parameter constraints on co-occurrence neighbourhoods; we design structurally distinct indexing trees; and our index exploit non-hierarchical locality between cores. The detailed contributions of this work are as follows:

(1) **Problem formulation**: This study presents, to our knowledge, the first index-based framework for efficient discovery of co-occurrence based cohesive subgraphs in hypergraphs.
(2) **Indexing algorithms**: We propose three new indexing techniques that mitigate excessive memory consumption by exploiting the inherent locality of hypergraph core structures, and further incorporate cross-parameter locality beyond hierarchical compression to enhance query efficiency.
(3) **Experimental evaluation**: We conduct extensive experiments on multiple real-world datasets, demonstrating the effectiveness of our proposed indexing structures and query algorithms.

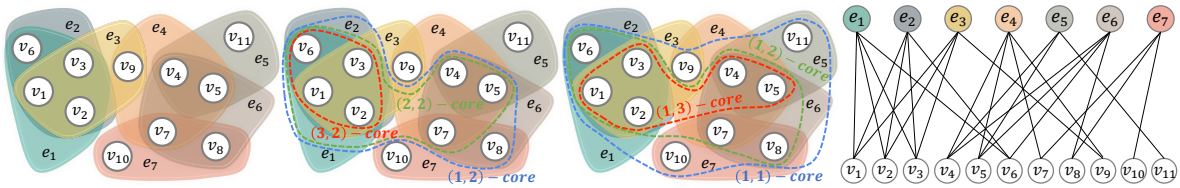
Organisation. The following sections of this paper are structured as follows: Section 2 introduces the key notations and definitions related to the (k, g)-core model. Section 3 describes the proposed index structures and research questions. In Section 4, we present our indexing model and algorithms, along with theoretical analysis. Section 5 provides empirical results from experiments on various real-world hypergraph datasets. Related work is discussed in Section 6, and the conclusion and the future work are presented in Section 7.

2 PRELIMINARIES

A hypergraph is formally represented as $G = (V, E)$, where V denotes the set of nodes and E denotes the set of hyperedges. In this paper, we assume G to be undirected and unweighted. For any subset $H \subseteq V$, we denote the induced subhypergraph as $G[H] = (H, E[H])$ where $E[H] = \{e \cap H \mid e \in E \wedge e \cap H \neq \emptyset\}$. We also define $B = \sum_{e \in E} |e|$ as the total cardinality of the hypergraph, and $|e^*| = \max_{e \in E} |e|$ as the largest cardinality among all hyperedges.

Within hypergraph terminology, it is essential to understand the definitions of fundamental terms such as "degree", "neighbours", and "cardinality", especially when contrasted with their traditional graph-theoretic counterparts.

- Degree: The *degree* of a node is defined as the number of hyperedges that include the node. This significantly differs from a traditional graph, in which the degree simply refers to the number of edges incident to the node.
- Neighbours: The *neighbours* of a node are all other nodes that are connected to it via shared hyperedges. This contrasts with



(a) A hypergraph **(b) Shell structure anchoring g** **(c) Shell structure anchoring k** **(d) Bipartite form of the graph**

Figure 2: Illustrative example of core structure patterns in a hypergraph

traditional graphs, where neighbours are only the nodes directly connected by an edge.

- Cardinality: The *cardinality* of a hyperedge indicates the number of nodes it contains. This concept is specific to hypergraphs, as edges in traditional graphs connect exactly two nodes, thereby fixing their cardinality at two.

DEFINITION 1. *((k,g)-core [22]). Given a hypergraph G, k, and g, (k,g)-core is the maximal set of nodes in which each node has at least k neighbours appearing in at least g hyperedges in an induced subhypergraph by the set of nodes.*

Based on this definition, we establish two fundamental properties of the (k,g)-core.

PROPERTY 1. *For $k' \leq k$ and $g' \leq g$, (k,g)-core $\subseteq$ (k',g')-core.*

PROOF. Assume there exists $v \in (k,g)$-core but $v \notin (k',g')$-core. Since v satisfies the condition, and $k' \leq k$, $g' \leq g$, it must satisfy the weaker (k',g') condition. This contradicts the assumption that $v \notin (k',g')$-core. Hence, (k,g)-core $\subseteq (k',g')$-core. □

PROPERTY 2. *(k,g)-core is unique.*

PROOF. Suppose that there exist two distinct (k,g)-cores H_1 and H_2. Based on the definition, both are maximal sets of nodes satisfying the (k,g) condition. Now consider their union $H' = H_1 \cup H_2$. Every node in H' satisfies the (k,g) condition within the induced subhypergraph of H'. Since $H_1 \subsetneq H'$ and $H_2 \subsetneq H'$, this contradicts the maximality requirement of H_1 and H_2. Therefore, the (k,g)-core is unique. □

We next define several notions of coreness that are utilised to build indexing structures.

DEFINITION 2. *(k-coreness / g-coreness). Given a hypergraph G and a fixed k, the k-coreness of a node x in G, denoted as $c^k(x,G)$, is the maximum g' such that x belongs to the (k,g')-core but not to the $(k,g'+1)$-core. Similarly, for a given g, the g-coreness of x, denoted as $c^g(x,G)$, is the maximum k' such that x is in the (k',g)-core but not in the $(k'+1,g)$-core.*

These single-parameter coreness capture the maximal participation of a node along one parameter side. However, to fully characterise a node position within the hierarchical structure of (k,g)-cores, we require a more comprehensive measure that accounts for both parameters simultaneously. It motivates the introduction of a joint coreness concept that captures the whole structural role of each node.

DEFINITION 3. *((k,g)-coreness). Given a hypergraph $G = (V,E)$ and a node $x \in G$, the (k,g)-coreness of x refers to the pairs of (k,g) values such that x belongs to the (k,g)-core but not to any (k',g)-core or (k,g')-core where $k' > k$ and $g' > g$.*

The k-coreness of a node x indicates the maximum number of hyperedges in which x co-occurs with at least k neighbours. Conversely, the g-coreness of a node x captures the maximum number of neighbours that co-occur with x in at least g hyperedges. Furthermore, the (k,g)-coreness jointly characterises the importance of a node by incorporating both its local connectivity and its interaction strength. Note that, unlike k-coreness or g-coreness where a node has a unique value for a given g or k, a node may possess multiple (k,g)-coreness values such as $(1,3)$, $(2,2)$, and $(3,1)$ simultaneously.

EXAMPLE 1. *Figure 2 illustrates an example hypergraph, its shell structures, and the corresponding hyperedge-node bipartite representation. Figure 2b shows the shell structure of the hypergraph G from Figure 2a when $g = 2$. The set of nodes $\{v_1, v_2, v_3, v_6\}$ (in red line) is included in all of the $(1,2)$, $(2,2)$, and $(3,2)$-cores. Hence, by the aforementioned definition, their g-coreness value with respect to $g = 2$, denoted as $c^2(v_i, G)$ for $i \in \{1,2,3,6\}$, is 3. Furthermore, in Figure 2c, the nodes $\{v_1, \ldots, v_5\}$ (in red line) form the $(1,3)$-core. Each node in this set has at least 1 neighbour with whom it co-occurs in at least 3 hyperedges. For instance, nodes v_1, v_2, v_3 appear together in e_1, e_2, and e_3, while v_4, v_5 co-occur in e_4, e_5, and e_6.*

Peeling algorithm for (k,g)-core. The peeling algorithm in [22] identifies the (k,g)-core by first computing all neighbours and then iteratively removing nodes that violate the core condition. While computing all neighbours initially can improve efficiency, this method requires $O(|V|^2)$ memory to store neighbour structures. To address this, we propose a memory-efficient variant that computes neighbours on the fly, reducing space to $O(|V|)$ with an increased time complexity of $O(|V| \cdot B \cdot |e^*|)$ where $B = \sum_{e \in E} |e|$ and $|e^*| = \max_{e \in E} |e|$. Since hyperedge sizes often follow a power-law distribution [16, 26], this trade-off is effective in practice. We denote the runtime of the peeling algorithm as $O(P)$ to reflect its adaptiveness. Detailed analysis on the algorithm can be checked in the online appendix [24].

3 PROBLEM STATEMENT

While certain structural properties of the (k,g)-core have been identified and studied in prior work, selecting appropriate user-defined parameters k and g remains a challenging and unresolved problem, as discussed earlier in Section 1. Therefore, in this paper, we aim to address this practical challenge by enabling end-users to adjust these parameters dynamically within online time in order to obtain the desired cohesive subgraph interactively and efficiently. The central research question and our solution are as follows:

Research Question. Given a hypergraph G, is it feasible for a user to dynamically change the k and g values and promptly obtain the corresponding (k,g)-core without recomputation?

Answer. To address above question, we aim to develop two approaches, each designed with a different strategic focus: (1) fast

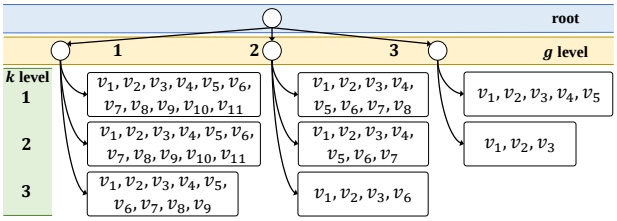


Figure 3: Naïve indexing tree structure

query processing time for responsiveness, and (2) compact space usage for scalability.

(1) **Naïve Indexing Approach (Naïve):** This method directly stores the results of the (k, g)-core for every possible valid combination of k and g in advance. As a result, end-users can access the desired (k, g)-core almost instantly without performing any complex query operations. This approach is particularly effective for handling small-sized datasets, where memory cost is not a critical constraint.

(2) **Locality-based Space Efficient Approach:** This method prioritises memory efficiency while maintaining flexible query access. To achieve this, we introduce several indexing strategies based on structural locality principles:
- *Horizontal Locality-based Indexing (LSE^H):* Leverages the inherent hierarchical relationships among (k, g)-cores with fixed g values to minimise redundant data storage.
- *Vertical Locality-based Indexing (LSE^{HV}):* Exploits the vertical core hierarchy across increasing g values to reduce duplication of node inclusion information.
- *Diagonal Locality-based Indexing (LSE^{HVD}):* Captures latent structural similarities among non-hierarchically related (k, g)-cores—particularly diagonally adjacent ones—by introducing and maintaining hidden auxiliary nodes.

4 INDEXING TREE AND QUERY PROCESSING

In this section, we present indexing algorithms designed for efficiently querying all (k, g)-cores in a flexible and scalable manner. We first introduce the Naïve Indexing Approach (Naïve), which prioritises query processing performance over memory efficiency by storing precomputed results. Due to its excessive memory usage, we then introduce Locality-based Space-Efficient Indexing approaches (LSEs), which focus on optimising memory usage while still maintaining reasonably efficient query processing times.

Indexing tree structure. Figure 3 illustrates the layered and hierarchical design of the proposed indexing structure. This structure is represented as a rooted tree of height 2 with three layers. At the root, the edges correspond to different values of g, forming the first layer of categorisation. At the g-level, the edges that connect nodes from this level to the k-level represent a specific k value, enabling efficient traversal for given parameter combinations. This structure provides an organised and compact representation of the various valid combinations of the (k, g) values. The k-level consists of (k, g)-leaf nodes, which store the actual node sets that are part of the corresponding (k, g)-core. The order of g and k in the tree structure is configurable—it can be determined by application context or user preference, and notably, it can be easily reversed without additional structural cost.

4.1 Naïve Indexing Approach (Naïve)

To support fast retrieval of any (k, g)-core, we propose a straightforward method called the Naïve Indexing Approach. It constructs an indexing tree that enables direct access to all (k, g)-cores.

4.1.1 *Indexing framework.*

Indexing tree construction. The construction of the Naïve indexing tree involves computing all possible (k, g)-cores in the given hypergraph. Each computed core is stored in its corresponding (k, g)-leaf node within the tree structure.

Leaf node. Each (k, g)-leaf node contains the set of nodes that belong to the associated (k, g)-core. These leaf nodes serve as the terminal storage units for the precomputed core information.

Query processing. Querying the Naïve indexing tree simply involves locating the appropriate (k, g)-leaf by traversing the root to the g-level and then to the k-level. The target leaf node directly provides all nodes in the specified (k, g)-core.

4.1.2 *Complexity analysis.*

Construction time complexity. Let $O(P)$[1] denotes the time complexity of the (k, g)-core peeling algorithm. For a specific value of g', let $k^*_{g'}$ denote the maximum k for that given g', implying that there is no $(k^*_{g'} + 1, g')$-core. To enumerate all leaf nodes of a branch corresponding to g', it is sufficient to compute only the $(k^*_{g'}, g')$-core since the computation of the $(k'_{g'}, g')$-core, where $k'_{g'} < k^*_{g'}$, is inherently part of the intermediate results obtained during the peeling procedure to get the $(k^*_{g'}, g')$-core. Thus, the time complexity to construct a Naïve indexing tree is $O(g^* \cdot P)$, where g^* is the maximum value of g in the hypergraph.

Query processing time complexity. Since each (k, g)-leaf node directly stores the nodes of the corresponding (k, g)-core, a query can be answered in $O(1)$ time using the Naïve indexing tree.

Space complexity. In the worst case, each (k, g)-leaf node may include all nodes in V, resulting in a total space usage of $O(k^* \cdot g^* \cdot |V|)$, where k^*, g^* are the maximum k, g value in the hypergraph.

EXAMPLE 2. *Consider the Naïve indexing tree illustrated in Figure 3, which is constructed from the hypergraph shown in Figure 2a. This indexing tree consists of eight (k, g)-leaf nodes, each corresponding to a unique combination of k and g values. Each leaf node stores the nodes in the associated (k, g)-core. Notably, some nodes appear frequently across multiple leaf nodes; for example, node v_1 appears in all of them.*

This observation highlights a key trade-off in the Naïve indexing technique: it offers optimal query processing time, i.e., $O(1)$, by directly storing all (k, g)-cores. However, it incurs a space complexity of $O(k^* \cdot g^* \cdot |V|)$, which becomes impractical to handle large-scale real-world hypergraphs due to excessive memory consumption. To mitigate this space overhead while preserving reasonable query performance, we next introduce three indexing techniques that encode core information by leveraging different types of locality observed in the (k, g)-core structure.

4.2 Horizontal Locality-based Indexing(LSE^H)

In this section, we present Horizontal Locality-based Indexing, which aims to reduce space complexity by utilising the hierarchical characteristics of the (k, g)-core structure. The term "horizontal" refers to the use of the k-level hierarchy within each g-branch in the indexing tree.

[1]The (k, g)-core peeling algorithm forms the basis of index technique. For simplicity and to facilitate analysis, we represent the complexity of the algorithm as $O(P)$.

Algorithm 1: `enum_h`: Enum shell structures by fixing g

Input: Hypergraph $G = (V, E)$, parameter g
Output: $\{(k, g)\text{-core} \mid k \geq 1 \text{ and } (k, g)\text{-core} \neq \emptyset\}$

```
1  prev ← ∅ , H ← V;
2  for k' ← 1 to |V| do
3      if |H| ≤ k' then
4          break;
5      while true do
6          changed ← false;
7          foreach v ∈ H do
8              N(v) ← {(w, c(v, w)) | w ∈ V, and c(v, w) ≥ g};
9              if |N(v)| < k' then
10                 H ← H \ {v};
11                 changed ← true;
12         if changed = false then
13             if prev ≠ ∅ then
14                 S.add(prev \ H);
15             prev ← H;
16             break;
17 if prev ≠ ∅ then
18     S.add(prev);
19 return S
```

Algorithm 2: Horizontal Locality-based indexing

Input: Hypergraph $G = (V, E)$
Output: Indexing tree T

```
1  T ← treeInit();
2  for g' ← 1 to g* do
3      k' ← 1;
4      S ← enum_h(G, g');
5      if |S| = ∅ then
6          break;
7      prev ← ∅;
8      for s ∈ S do
9          u ← insertLeafNode(T, k'++, g', s);
10         if prev ≠ ∅ then
11             prev.next ← u;
12         prev ← u;
13 return T
```

4.2.1 Indexing framework.

Indexing tree construction. The construction process for the LSE^H indexing tree is described in Algorithm 1 and Algorithm 2. The enumeration process (Algorithm 1) iteratively increases k' for a fixed value of g to identify the corresponding (k', g)-cores. If the number of remaining nodes becomes less than k', no further (k', g)-cores can be obtained for that g, and the search terminates (Lines 1–4). Otherwise, for each node $v \in V$, the algorithm identifies neighbour nodes w such that v and w appear together in at least g hyperedges, and checks whether the number of such neighbours is at least k'. Nodes that do not satisfy this condition are removed through peeling process (Lines 5-11). In addition, it removes redundancies between subsequent (k, g)-cores, such as the (k', g)-core and the $(k'+1, g)$-core (Lines 13-15). Based on this process, in Algorithm 2, we iterate over g' from 1 to the maximum value g^* and compute the shell structure for each g' (Lines 1-4). Since the enumeration identifies the g-coreness for all nodes, only the nodes satisfying $c^{g'}(v, G) = k'$ are stored in the corresponding (k', g')-leaf node (Lines 8-9). These leaf nodes are connected via "next links" (Lines 10-12). As a result, the leaf nodes sharing the same parent in the g-level do not contain redundant nodes.

Leaf node. Each leaf node contains both a specific value and a link to the next leaf node in the sequence. The composition of a leaf node in this structure is detailed as follows:

- *Value:* Each (k, g)-leaf node in the indexing tree contains a set of nodes, all of which share the same g-coreness, denoted by the value k. This implies that nodes having identical g-coreness are stored in the same leaf node.
- *Link:* Given a specific g value, a leaf node representing a certain g-coreness of exactly k is linked to the subsequent leaf node that represents g-coreness of exactly $k + 1$. This edge is referred to as the *next link*, and this linked-list structure is consistently repeated across every branch corresponding to different g values, thereby maintaining a uniform and systematic connection pattern within the indexing tree.

Query processing. To process a query $Q = (k, g)$ with the indexing tree, the following procedure is employed to retrieve the (k, g)-core: Initially, we find the (k, g)-leaf node. From this leaf node, it iteratively traverses subsequent leaf nodes through *next link* until reaching the terminal leaf node, which is not pointing to a further leaf node. The (k, g)-core is retrieved by aggregating all nodes encountered across these traversed leaf nodes.

4.2.2 Complexity analysis.

Construction time complexity. The time complexity for constructing the LSE^H indexing tree is $O(g^* \cdot (P + k^* \cdot |V|))$, where $O(P)$ denotes the time complexity of the (k, g)-core peeling algorithm. Compared to the Naïve indexing tree construction, this process includes an additional step: performing a set difference operation between pairs of linked leaf nodes, which takes $O(|V|)$ time in the worst case for each pair. Since at most k^* such operations can occur for each $g > 1$, the additional overhead across all g layers remains bounded by $O(g^* \cdot k^* \cdot |V|)$.

Query processing time complexity. Query processing involves traversing the leaf nodes in the indexing tree T, starting from the (k, g)-leaf node and continuing until the terminal leaf node is reached. Thus, the overall time complexity for retrieving the (k, g)-core is $O(k^* \cdot |V|)$, which clearly reflects the linear dependency on the number of nodes and the maximum k value.

Space complexity. The LSE^H indexing technique can significantly reduce space complexity by storing each node in the hypergraph only once for each g value, unlike the Naïve indexing approach which may store redundant node information. This efficiency is achieved through a horizontal connection of leaf nodes and the removal of redundancies across different k values for the same g. Consequently, the space complexity is primarily determined by the number of distinct nodes for each g value, leading to a complexity of $O(g^* \cdot |V|)$.

EXAMPLE 3. *Figure 4 presents the LSE^H indexing tree applied to the hypergraph G presented in Figure 2a. In this representation, nodes $\{v_1, \ldots, v_9\}$ are categorised within the $(3, 1)$-leaf. This indicates that these nodes are included in both the $(1, 1)$-core and $(2, 1)$-core, i.e., $\{v_1, \ldots, v_9\} = (3, 1)$-core $\subseteq (2, 1)$-core $\subseteq (1, 1)$-core. When processing a query $Q = (2, 1)$, the method involves traversing from the $(2, 1)$-leaf node to $(3, 1)$-leaf node through the next link, subsequently aggregating the nodes from these leaf nodes to obtain the $(2, 1)$-core. Note that a node v may appear in multiple leaf nodes across different g branches if its k-coreness is greater than 1. For example, nodes v_1, v_2, and v_3 are present in all g branches.*

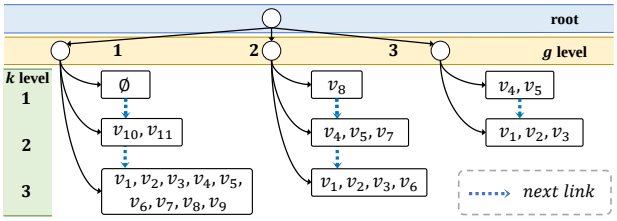


Figure 4: LSE^H indexing tree structure

4.3 Vertical Locality-based Indexing(LSE^{HV})

In this section, we introduce the Vertical Locality-based Indexing technique, developed to improve space complexity by simultaneously considering both k and g parameters. Unlike LSE^H, which focuses on overlaps within the same g value, LSE^{HV} handles overlaps across different k and g values at the same time. For instance, while LSE^H eliminates duplicate nodes between $(1,2)$-core, $(2,2)$-core, and $(3,2)$-core, it does not consider overlaps between cores like $(3,1)$-core and $(3,2)$-core. The LSE^{HV} indexing technique, therefore, has additional links between leaf nodes of the same k value, preserving efficient query processing by accounting for duplicate relationships across both k and g values.

4.3.1 *Indexing framework.*

Indexing tree construction. The construction process for the LSE^{HV} indexing tree is described in Algorithm 3. It begins based on the LSE^H indexing tree as a preliminary step (Lines 1–12). Then, for each g' from 1 to $g^* - 1$, where g^* is the maximum g value in the hypergraph (Line 13), the algorithm performs set difference operations between the (k', g')- and $(k', g' + 1)$-leaf nodes for all k' up to $k^*_{(g'+1)}$, where k^*_g denotes the maximum k value for a given g (Lines 14–17). These operations establish "jump links" between adjacent leaf nodes across g-level layers, as illustrated in Figure 5. As with the LSE^H indexing tree, the number of nodes in each g-layer branch remains bounded by $|V|$.

Leaf node. In the Vertical Locality-based Indexing (LSE^{HV}), each leaf node contains a specific value and two distinct links. The components of a leaf node are as follows:

- *Value:* In the LSE^{HV} indexing tree, each (k,g)-leaf node comprises a unique set of nodes. These nodes are characterised by having a k-coreness of exactly g and a g-coreness of exactly k. This means that the (k,g)-coreness of any node in a (k,g)-leaf node includes the pair of values (k,g).
- *Link:* In the LSE^{HV} indexing tree, links are categorised into two distinct types: *next link* and *jump link*.
  (1) Next Link: The next link, the same as in the LSE^H indexing tree, connects to the subsequent leaf node in the sequence. Specifically, a (k,g)-leaf node is linked to the $(k+1,g)$-leaf node. This link is based on the g-coreness, determining the exact position of a node within the indexing tree.
  (2) Jump Link: The jump link connects to the next leaf node fixing the k value. For instance, a (k,g)-leaf node is linked to the $(k,g+1)$-leaf node. Based on the k-coreness, it guides the location of a node within the indexing tree.

Query processing. To process a query $Q = (k,g)$ using the LSE^{HV} indexing tree, we employ a specific procedure to accurately find the (k,g)-core. The first step is to find the corresponding (k,g)-leaf node in the indexing tree. Starting from this node, the process involves iterative traversal of subsequent leaf nodes along the jump link until the terminal leaf node is reached. Subsequently, for each of these leaf nodes, traversal continues along the next link until the terminal leaf node is encountered. The (k,g)-core is then obtained by aggregating all nodes during these traversals across the respective leaf nodes.

Algorithm 3: Vertical Locality-based indexing

Input: Hypergraph $G = (V, E)$
Output: Indexing tree T
`/* Lines 1-12 in Algorithm 2 */`
13 **for** $g' \leftarrow 1$ **to** $g^* - 1$ **do**
14 **for** $k' \leftarrow 1$ **to** $k^*_{(g'+1)}$ **do**
15 `Leaf(k',g').jump` $\leftarrow$ `Leaf(k',g'+1);`
16 $s \leftarrow$ `Leaf(k',g')` $\setminus$ `Leaf(k',g'+1);`
17 `updateLeafNode(T,k',g',s);`
18 **return** T

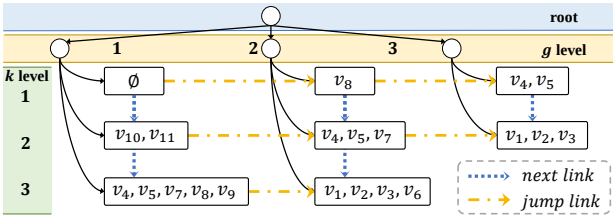


Figure 5: LSE^{HV} indexing tree structure

4.3.2 *Complexity analysis.*

Construction time complexity. The LSE^{HV} algorithm extends the LSE^H approach by visiting the $(g^* - 1)$ children of the root node and performing additional set difference operations between k^*_g leaf nodes, where k^*_g denotes the maximum k value for a given g. The complexity of these operations is bounded by $O(k^* \cdot |V|)$. Therefore, the overall time complexity for constructing the LSE^{HV} indexing tree is $O(g^* \cdot (P + k^* \cdot |V|))$, where $O(P)$ denotes the time complexity of the (k,g)-core peeling algorithm.

Query processing complexity. In the worst case, LSE^{HV} requires traversing all leaf nodes while aggregating the nodes. Hence, the time complexity for query processing is $O(k^* \cdot g^* \cdot |V|)$.

Space complexity. The space complexity of LSE^{HV} indexing tree is equivalent to that of the LSE^H indexing tree, which is $O(g^* \cdot |V|)$. This is because, in the worst case, nodes may not be stored adjacently within the indexing tree. For instance, if a node v_i is present in the (k,g)-leaf node and also belongs to the $(k-1,g+1)$-core, duplication of nodes within the indexing tree is inevitable.

EXAMPLE 4. *Figure 5 shows the LSE^{HV} indexing tree for the hypergraph G from Figure 2a. It shows that the LSE^{HV} indexing tree reduces redundancy by compressing vertical overlaps. For example, the $(3,1)$-leaf in the LSE^{HV} indexing tree contains only $\{v_4, v_5, v_7, v_8, v_9\}$, as LSE^{HV} removes duplicate nodes from the same k-level branches. To process the query $Q = (2,2)$, it finds the $(2,2)$-leaf and uses the next links to aggregate nodes from leaves such as $(3,2)$. Then, it follows the jump links to include nodes from leaves like $(2,3)$. As a result, it retrieves nodes $\{v_1, \ldots, v_7\}$, corresponding to the $(2,2)$-core.*

4.4 Diagonal Locality-based Indexing(LSE^{HVD})

In this section, we present the Diagonal Locality-based Indexing technique. Beyond removing redundancies in hierarchical (k,g)-cores, it captures the locality among non-hierarchical (k,g)-cores, focusing on the relationships between diagonally placed leaf nodes such as $(k-1,g)$-leaf node and $(k,g-1)$-leaf node.

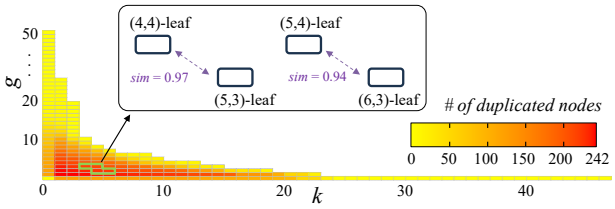


Figure 6: Number of nodes in diagonal leaf nodes

Rationale of LSE^{HVD} approach. The LSE^{HVD} technique is motivated by the observation that strong locality exists even among non-hierarchical cores within the Naïve indexing tree. To validate this, we analysed the *Contact* dataset [3], measuring the number of common nodes between diagonally adjacent leaf nodes (e.g., $(k-1,g)$ and $(k,g-1)$). Using Jaccard similarity, we quantified the overlap and averaged it across cases where a leaf node has two diagonally adjacent nodes. Figure 6 illustrates that a significant overlap exists between these diagonally adjacent leaf nodes, indicating strong diagonal locality in real-world networks. This finding supports the design of the LSE^{HVD} technique, which reduces redundancy by identifying and relocating overlapping nodes into auxiliary nodes. These auxiliary nodes may themselves be diagonally adjacent and recursively optimised. To achieve this, the LSE^{HVD} indexing tree is built atop the LSE^{HV} indexing tree, further compressing the structure by exploiting diagonal locality.

Auxiliary nodes. Auxiliary nodes serve a key role in the LSE^{HVD} indexing tree. For each (k,g)-leaf node where $k>1$ and $g>1$, a corresponding auxiliary node—denoted as the (k,g)-aux node—is defined. Each auxiliary node consists of key-value pairs: the key represents a depth level, and the value is a set of nodes. Depth indicates the degree of hierarchical commonality. At depth $d=1$, the value represents nodes shared between the $(k-1,g)$ and $(k,g-1)$ leaf nodes of the LSE^{HV} indexing tree. At depth $d=2$, the value includes nodes that appear at depth 1 in both $(k-1,g)$ and $(k,g-1)$-aux nodes—implying shared presence in $(k-2,g)$, $(k-1,g-1)$, and $(k,g-2)$-leaf nodes. This depth-aware key-value design enables layered redundancy elimination while maintaining a compact representation.

LEMMA 1. *If a node y exists in a (k,g)-leaf node, then it cannot coexist in the corresponding (k,g)-aux node.*

PROOF. The existence of node y in a (k,g)-leaf node implies that the k-coreness of the node is g and its g-coreness is k. However, if the same node y also exists in the (k,g)-aux node, it implies that the k-coreness of y is less than g, or its g-coreness is less than k. This contradicts the conditions for belonging to the (k,g)-leaf node. Therefore, it is not possible for a node to coexist in both a (k,g)-leaf node and its corresponding (k,g)-aux node. □

THEOREM 1. *In any (k,g)-aux node, no node appears at more than one depth.*

PROOF. Suppose that in a (k,g)-aux node of the LSE^{HVD} indexing tree, where $k>1$ and $g>1$, there exist sets of nodes at depths d' and d'', and a node y appears in both sets. For simplicity, we assume $d'<d''$. First, let consider the existence of node y at depth d''. This necessitates its existence in the leaf nodes $(k-d'',g)$-leaf node, $(k-(d''-1),g-1)$-leaf, $\cdots$, $(k,g-d'')$-leaf node. Consequently, node y must also be present in the auxiliary nodes $(k-d^*,g)$-aux node, $(k-(d^*-1),g-1)$-aux node, $\cdots$, and $(k,g-d^*)$-aux node where $d^*=d''-1$. Since node y is in (k,g)-aux node, it indicates that node y iteratively appears in all the intermediate auxiliary nodes, and finally compressed in (k,g)-aux node. Since node y is at depth d', it indicates that node y appears in the leaf nodes $(k-d',g)$-leaf node, $(k-(d'-1),g-1)$-leaf node, $\cdots$, $(k,g-d')$-leaf node. This is clearly contradictory, since node y cannot appear at (k,g)-leaf node and (k,g)-aux node simultaneously as we have checked in Lemma 1. Thus, it is proven that in a (k,g)-aux node, There cannot be a node stored redundantly at two different depths. □

Algorithm 4: Diagonal Locality-based indexing

```
Input: Hypergrpah G = (V, E)
Output: Indexing tree T
/* Lines 1-17 in Algorithm 3                                   */
18 A ← ∅;
19 for g ← 1 to g* do
20     cur ← Leaf(1, g);
21     while cur.next do
22         if exist(Leaf(cur.k, g + 1)) &
             exist(Leaf(cur.k + 1, g)) then
23             cj ← cur.jump;
24             cn ← cur.next;
25             I ← cj ∩ cn;
26             if exist(Leaf(cur.k + 1, g + 1)) then
27                 A ← Leaf(k + 1, g + 1);
28             else
29                 A ← emptyLeaf();
30                 cj.next, cn.jump ← A;
31             A.put(makeAux(d = 1, I));
32             cn.removeAll(I);
33             for d' ∈ dep(Aux(cn)) ∩ dep(Aux(cj)) do
34                 I' ← Aux(cn).get(d') ∩
                     Aux(cj).get(d');
35                 A.put(makeAux(d = d' + 1, I'));
36                 Aux(cn).get(d').removeAll(I');
37         cn.removeAll(Aux(cn.next).get(1));
38         else
39             while exist(cn.next) do
40                 for d' ∈ dep(Aux(cn)) do
41                     s ← Aux(cn).get(d') \
                         Aux(cn.next).get(d' + 1);
42                     Aux(cn).get(d').removeAll(s);
43                 cn ← cn.next;
44         cur ← cn.next;
45 return T
```

COROLLARY 1. *In any auxiliary node (k,g)-aux, a node v appears at most once, at a single depth.*

This ensures that the use of auxiliary nodes does not increase the total number of stored nodes, resulting in a more compact structure than the LSE^{HV} indexing tree.

4.4.1 Indexing framework.

Indexing tree construction. Algorithm 4 outlines the construction process. It begins by creating the LSE^{HV} indexing tree (Lines 1–17). Next, starting from the $(1,1)$-leaf node, it checks for diagonal adjacency. If a pair of diagonally adjacent leaf nodes is found (e.g., $(k'+1,g')$ and $(k',g'+1)$), overlapping nodes are stored in the $(k'+1,g'+1)$-aux node at depth 1 (Lines 22–31). If the aux node does not exist, it is created. These redundant nodes are then removed from the original leaf nodes. Further, overlapping nodes at the same depth in auxiliary nodes are handled (Lines 32–36),

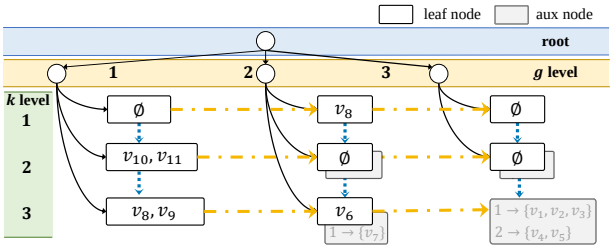


Figure 7: LSE^{HVD} indexing tree structure

followed by the elimination of any remaining redundancy between leaves and their auxiliary nodes (Line 37). If no further diagonally placed nodes exist, the algorithm traverses the next links for the current g value to eliminate redundancies between auxiliary nodes (Lines 39–43).

Leaf node. Each leaf node in the LSE^{HVD} indexing tree contains a set of nodes and links (next and jump). A distinctive feature is the auxiliary node structure, which stores overlapping nodes that appear in diagonally adjacent leaf nodes. As a result, these overlapping nodes are removed from the leaf nodes and maintained centrally in the appropriate auxiliary node.

Query processing. Given a query $Q = (k, g)$, the process first retrieves the corresponding leaf node. It then traverses all reachable leaf nodes using both jump and next links, aggregating their contents in an orderly manner. Unlike in the LSE^{HV} tree, the LSE^{HVD} variant also requires incorporating auxiliary nodes. Importantly, auxiliary nodes attached to the starting leaf are excluded. If an auxiliary node is encountered along a path, its values are aggregated using the path distance as a key. For example, if the (k, g) leaf node connects to $(k, g + 1)$ via a next link, the aggregation key is depth 1; if a jump link leads to $(k + 1, g + 1)$-aux, then both depths 1 and 2 are used. In other words, each (k, g)-leaf node only considers depths up to the minimum hops required to reach each reachable (k', g')-leaf or aux node.

4.4.2 *Complexity analysis.*

Construction time complexity. The LSE^{HVD} tree is constructed atop the LSE^{HV} indexing tree, with additional steps to identify and extract overlaps among diagonally adjacent nodes. This results in time complexity $O(g^* \cdot (P + k^* \cdot |V|))$, where P denotes the peeling procedure complexity.

Query processing time complexity. The time complexity for query processing in the LSE^{HVD} indexing tree is comparable to that of the LSE^{HV} indexing tree. While the presence of auxiliary nodes may incur additional operations, these do not impact the overall time complexity. Thus, the time complexity for query processing in the LSE^{HVD} indexing tree is still bounded by that of the LSE^{HV} indexing tree, specifically $O(k^* \cdot g^* \cdot |V|)$.

Space complexity. The space complexity of the LSE^{HVD} indexing tree is equivalent to that of the LSE^{H} indexing tree. However, in practice, it offers improved space efficiency when diagonal locality is prevalent in real-world graphs.

EXAMPLE 5. *Figure 7 illustrates the LSE^{HVD} indexing tree with auxiliary nodes, constructed based on the hypergraph in Figure 2a. Compared to the LSE^{HV} indexing tree shown in Figure 5, the LSE^{HVD} indexing tree effectively reduces redundancy by storing shared nodes between diagonally adjacent leaf nodes only once in appropriate auxiliary nodes. For example, the nodes v_4 and v_5, which appear in multiple leaf nodes—specifically $(1, 3)$, $(2, 2)$, and $(3, 1)$—are consolidated into a single auxiliary, the $(3, 3)$-aux node at depth 2. As a result, redundant storage of nodes across leaf layers is significantly reduced. For querying, for instance, $Q = (3, 2)$, we start at the $(3, 2)$-leaf node, skip auxiliary nodes of the starting leaf node, and move through the jump nodes to the terminal auxiliary node. After one move through the jump link, in the $(3, 3)$-aux node, we store only the nodes with depth 1, resulting in the nodes $\{v_1, v_2, v_3, v_6\}$.*

Summary of Algorithms. Each indexing technique is carefully designed to minimise memory usage while ensuring efficient query processing. Table 1 provides a comprehensive efficiency analysis of the algorithms, including their time and space complexities. The table also further details the leaf node structures, types of links, and the presence of auxiliary nodes for each resulting index from the algorithm. We use a star notation ($\star$) to indicate efficiency levels, where a higher number of stars represents greater efficiency. Overall, the table clearly highlights a practical trade-off between query processing efficiency and space efficiency. It is also worth noting that even with identical complexity, the actual performance can vary significantly, as complexity analysis considers extreme scenarios that may not fully reflect the diverse characteristics of real-world hypergraphs.

Beyond efficiency, our indexing structures also provide structural insights into hypergraphs. Leaf nodes in LSE^{H} and LSE^{HV} trees correspond to g-coreness and (k, g)-coreness, offering quantitative measures of node cohesion [4, 25, 34]. Moreover, LSE^{HVD} indexing captures non-hierarchical relations overlooked by previous methods: its auxiliary nodes consolidate common nodes across diagonally related cores. Nodes placed at deeper auxiliary levels frequently occur in multiple strong cores and thus act as bridges between them, whose removal may compromise network stability [40, 44]. This demonstrates that our indexing structures are not only memory-efficient but also support meaningful structural interpretation of hypergraph topology.

5 EXPERIMENTS

5.1 Experiment Setup

We conducted extensive experiments to evaluate the performance of our indexing techniques, guided by a set of key evaluation questions (EQs) across diverse scenarios.

- **EQ1. Efficiency of index construction and query processing:** How efficient are the proposed algorithms in terms of index construction and query processing time?
- **EQ2. Scalability:** How well does the proposed method scale on benchmark hypergraphs with increasing sizes?
- **EQ3. Effectiveness of the indexing technique:** To what extent does our indexing method reduce node duplication?
- **EQ4. Structural insights from the indexing tree:** What structural patterns can be observed from the constructed indexing trees?
- **EQ5. Effect on k^* and g^*:** How do the maximum values k^* and g^* influence the index construction time and memory usage?
- **EQ6. (k, g)-core effectiveness:** How effective is the (k, g)-core in identifying cohesive and structurally significant subgraphs?
- **EQ7. Case study on real data:** How effective is the proposed indexing framework in real-world scenarios?

5.2 Experimental Setting

Dataset. We used eight real-world datasets, along with four synthetic hypergraphs generated by HyperFF [26], using a burning

Table 1: Summary of the algorithms

Algorithms	Querying efficiency		Space efficiency		Value of (k,g)-leaf	Links		Aux
LSE^H	⋆⋆⋆⋆	$O(k^*\|V\|)$	⋆⋆⋆	$O(g^*\|V\|)$	Nodes with same g-coreness	Next link		×
LSE^{HV}	⋆⋆⋆	$O(k^*g^*\|V\|)$	⋆⋆⋆⋆	$O(g^*\|V\|)$	Nodes with same (k,g)-coreness	Next link	Jump link	×
LSE^{HVD}	⋆⋆⋆	$O(k^*g^*\|V\|)$	⋆⋆⋆⋆⋆	$O(g^*\|V\|)$	Aux : diagonally adjacent common nodes Leaf: LSE^{HV} (k,g)-leaf \ Aux	Next link	Jump link	○

Table 2: Real-world dataset

Dataset	$\|V\|$	$\|E\|$	$\mu(N(.))$	k^*	g^*
Contact (**CT**)	242	12,704	68.74	47	54
Congress (**CG**)	1,718	83,105	494.68	368	1003
Enron (**E**)	4,423	5,734	25.35	40	392
Meetup (**M**)	24,115	11,027	65.27	121	250
Walmart (**W**)	88,860	69,906	5.18	127	729
Trivago (**T**)	172,738	233,202	12.68	84	63
DBLP (**D**)	1,836,596	2,170,260	9.05	279	221
AMiner (**A**)	27,850,748	17,120,546	8.38	610	730
HF1K	1,000	2,233	4.564	7	4
HF10K	10,000	22,395	4.601	8	4
HF100K	100,000	221,111	4.542	9	5
HF1M	1,000,000	2,210,517	4.541	11	6

parameter $p = 0.4$ and an expanding parameter $q = 0.05$, to comprehensively evaluate scalability. The statistics of the datasets are summarised in Table 2, where $\mu(N(.))$ denotes the average neighbour size of all nodes, and k^* and g^* indicate the maximum observed values of k and g, respectively. All real-world datasets used in this study are publicly available [3, 7, 11].

Algorithms. As no prior work directly addresses (k,g)-core decomposition with indexing, we include Bi-core index [31] as a comparative baseline, despite its different structural assumptions.

Experimental environment. We implemented the indexing tree construction and query processing algorithms in C++. All experiments were performed on a server equipped with an Intel Xeon 6248R processor and 256GB of RAM, running Ubuntu 20.04.

5.3 Experimental Result

Table 3: Index construction time (seconds)

Dataset	Naïve	LSE^H	LSE^{HV}	LSE^{HVD}
Contact	0.9775	1.2386	1.2388	1.2389
Congress	3869.56	5679.26	5679.27	5679.29
Enron	16.7516	17.4130	17.4146	17.4148
Meetup	145.685	289.034	289.040	289.048
Walmart	785.593	868.832	868.845	868.856
Trivago	70.868	163.813	163.845	163.871
DBLP	2218.23	2573.91	2574.52	2575.04
AMiner	38258.5	108,542	108,560	108,570

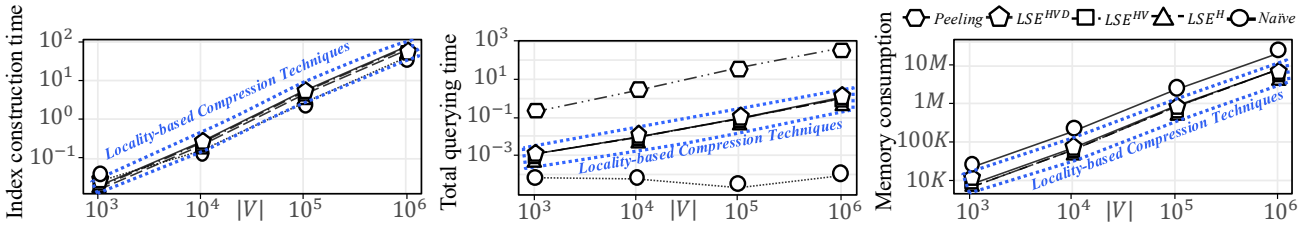
EQ1. Efficiency of index construction and query processing. We analysed the index construction and query processing times for each indexing technique. As shown in Table 3, the construction time increases slightly from LSE^H to LSE^{HV}, and then to LSE^{HVD}, due to the additional structural dependencies across indexing schemes. The Naïve indexing tree requires consistently less construction time. For query processing, we sorted all (k,g)-cores by size using the Naïve indexing tree and selected 100 queries corresponding to the 1^{st} to 100^{th} percentiles to report the total processing time. As presented in Table 4, while Naïve achieves constant query time by avoiding indexing tree traversal, query times gradually increase from LSE^H to LSE^{HVD}. On average, query processing on LSE^{HV} is 1.74× slower than LSE^H, and LSE^{HVD} is 1.56× slower than LSE^{HV}. Nevertheless, although LSE^{HVD} exhibits the longest query time among our indexing strategies, it still provides up to 3,600× speedup compared with the baseline peeling algorithm. These findings demonstrate that the proposed indexing strategies achieve efficient construction and querying, ensuring practical online query processing.

Table 4: Query processing time (seconds)

Dataset	Naïve	LSE^H	LSE^{HV}	LSE^{HVD}	Peeling
Contact	0.0003	0.0010	0.0015	0.0016	0.6970
Congress	0.0009	0.0112	0.0288	0.0774	108.43
Enron	0.0003	0.0011	0.0041	0.0085	2.0878
Meetup	0.0004	0.0100	0.0119	0.0159	50.382
Walmart	0.0003	0.0109	0.0128	0.0253	91.047
Trivago	0.0004	0.0327	0.0350	0.0431	82.783
DBLP	0.0004	0.3570	0.4653	0.4806	538.04
AMiner	0.0007	0.9210	1.2730	1.3795	17994.5

EQ2. Scalability. Figure 8 presents the scalability results of our algorithms on the synthetic hypergraphs introduced in Section 5.2. As shown in Figure 8a, for small-sized hypergraphs, the construction times across all index types remain comparable. The Naïve indexing tree requires slightly longer construction time than LSE because of the overhead of storing redundant nodes. As the data size increases, the differences in time complexity become more evident, with LSE indexing trees requiring more construction time. Nevertheless, all index construction methods demonstrate approximately linear scalability. For query evaluation, the k and g parameters were selected from (k,g)-cores corresponding to the 1^{st} to 10^{th} percentiles of core sizes, and we report the total time for all queries. Figure 8b shows that the query time of the Naïve indexing tree remains constant regardless of data size, which is explained by its $O(1)$ complexity. In contrast, all LSE indexing trees present nearly linear scalability. Regarding byte-memory usage, as presented in Figure 8c, memory efficiency follows the order of Naïve, LSE^H, LSE^{HV}, and LSE^{HVD}, with all methods exhibiting nearly linear growth patterns for similar network structures. These results indicate that our indexing techniques provide scalable indexing construction and query processing times with efficient memory utilisation.

EQ3. Effectiveness of indexing technique. To evaluate the effectiveness of our indexing techniques, we compared the memory usage and the number of nodes across four indexing trees with real-world datasets. As presented in Figure 9, we first observe that the Naïve indexing tree uses more memory than the actual data



(a) Index construction time (b) Total querying time (c) Memory usage

Figure 8: Scalability test

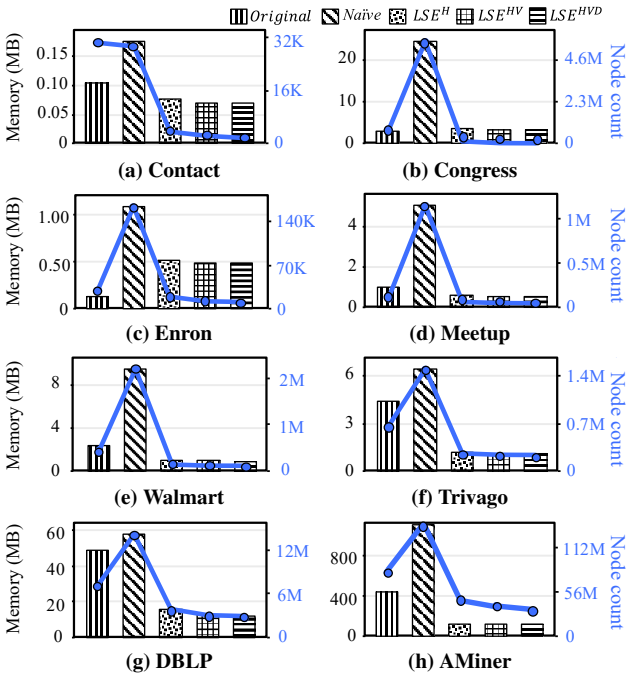


Figure 9: Memory usage & node size comparison

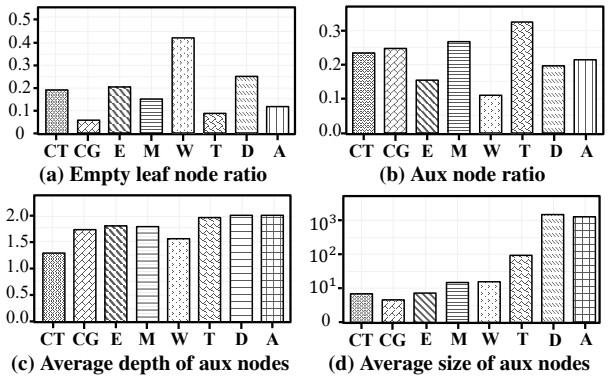
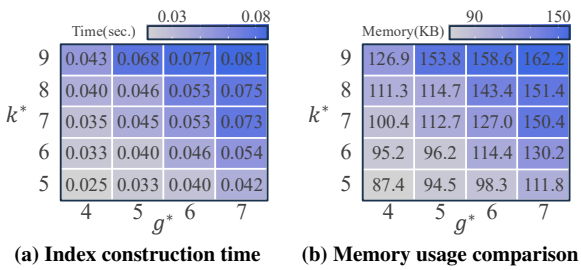


Figure 10: Leaf & aux node statistics

across all datasets and stores more nodes than those belonging to the actual data for all datasets except Contact data. This indicates that the Naïve indexing tree inefficiently stores a large number of redundant nodes. We also found that there is a significant decrease in node count when moving from the Naïve indexing tree to the LSE^{H} indexing tree, indicating that incorporating g-coreness effectively eliminates redundancy. The LSE^{HV} indexing tree exploits k-coreness to reduce both memory usage and node count retained in the LSE^{H} structure. Additionally, the LSE^{HVD} contributes an additional reduction. These results demonstrate that our indexing performs efficient compression with only a slight additional construction and querying time.

EQ4. Structural insights from the indexing tree. Figure 10 presents detailed statistics on leaf and auxiliary nodes after LSE^{HVD} indexing, including four metrics computed as follows:

- *Empty leaf node ratio* (Figure 10a) is calculated as the number of empty leaf nodes divided by the total number of leaf nodes, i.e., $\frac{\#\text{ empty leaf nodes}}{\#\text{ total leaf nodes}}$. On average, this ratio is around 19%. This result confirms that the proposed compression method effectively removes structural redundancies by exploiting horizontal, vertical, and diagonal localities.
- *Auxiliary node ratio* (Figure 10b) is the number of auxiliary nodes divided by the total number of leaf nodes, i.e., $\frac{\#,\text{auxiliary nodes}}{\#,\text{total leaf nodes}}$. In most datasets, this ratio exceeds 20%, which indicates that numerous structurally relevant nodes are shared across multiple non-nested (k, g)-cores. This clearly reflects the strong ability of our indexing tree in capturing diagonal locality that was largely overlooked by previous approaches.
- *Average depth of auxiliary nodes* (Figure 10c) measures the average of the maximum depths of all auxiliary nodes. Most datasets exhibit an average depth close to 2, which implies that localities between distant cores are frequently captured within the indexing tree. This result further suggests that our indexing tree effectively captures subtle non-hierarchical localities beyond directly adjacent cores.
- *Average size of auxiliary nodes* (Figure 10d) is defined as the total number of stored nodes in each auxiliary node divided by the number of auxiliary nodes. A large auxiliary node indicates strong locality across non-hierarchical cores, suggesting that dense interactions frequently occur around the node. These auxiliary nodes can thus serve as local hubs where such interactions are concentrated. For instance, in the Trivago dataset, relatively large auxiliary nodes further hint at the presence of particularly influential hub nodes that facilitate local connections.



(a) Index construction time (b) Memory usage comparison

Figure 11: Effect of k^* and g^*

EQ5. Effect on k^* and g^*. To analyse the impact of maximum k and g values (k^* and g^*) on the performance and memory usage of our algorithm, we generated 20 hypergraphs of size 1,000 using HyperFF [26], varying k^* from 4 to 7 and g^* from 5 to 9. We measured the running time and memory usage of constructing the LSE^{HVD} indexing tree on each dataset. As shown in Figure 11, both k^* and g^* exhibit a clear increasing trend in memory usage and runtime. The growth patterns are non-linear, and in our experimental setting, g^* increments tended to produce slightly steeper increases than k^* increments. Overall, these results indicate that algorithm performance is affected not only by network size but also its level of hierarchy, as larger k^* or g^* values naturally introduce heavier computational and memory demands.

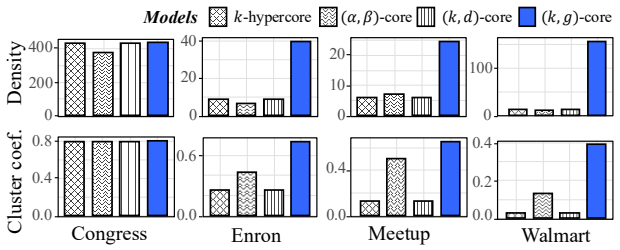


Figure 12: Cohesive subhypergraph model comparison

EQ6. (k,g)-core effectiveness. To evaluate the effectiveness of the (k,g)-core model beyond index efficiency, we compared it with three representative cohesive subgraph models in bipartite, and hypergraph: the k-hypercore [29], the (α,β)-core [14], and the (k,d)-core [3]. For a fair comparison, all parameters were fixed at 5, and the evaluation was conducted on four real-world datasets. The results in Figure 12 present that the (k,g)-core consistently outperforms the other models in terms of clustering coefficient and node density. In particular, it achieves markedly higher values in both metrics, reflecting its stronger ability to identify dense subgraphs and capture frequent co-occurrence patterns. These findings indicate that incorporating both neighbour size and co-occurrence enables the (k,g)-core to uncover structurally significant patterns in hypergraphs.

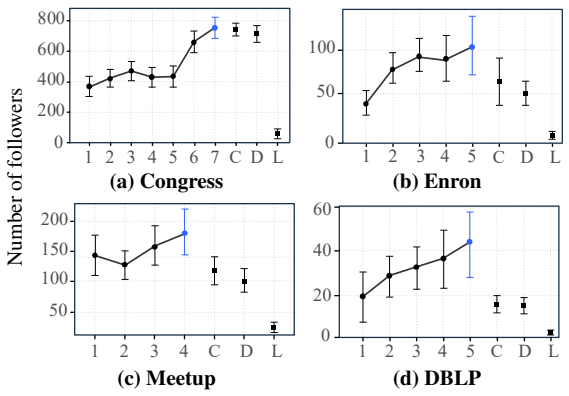


Figure 13: Case study - Resilience of nodes by depth

EQ7-1. Case study: LSE^{HVD} for network resilience analysis. Unlike earlier indexing trees that consider only hierarchical core structures, LSE^{HVD} approach organises non-hierarchical relationships through auxiliary nodes, where larger depths correspond to nodes distributed across multiple cores. To evaluate insights beyond memory efficiency, we conduct a follower analysis to examine network stability. Followers of a node v are nodes that drop out of their original cores once v is deleted, a metric used to measure stability [44]. We sampled 50 nodes from each depth level across four networks and compared their average follower counts against three baselines: (C) top 50 nodes by coreness, (D) top 50 nodes by degree, and (L) 50 random nodes only in leaf node. As shown in Figure 13, follower counts increase with depth, and the highest-depth nodes consistently yield more followers than all baselines. These findings demonstrate that LSE^{HVD} uncovers structural patterns overlooked by purely hierarchical analyses, extending its utility from memory optimisation to understanding network resilience.

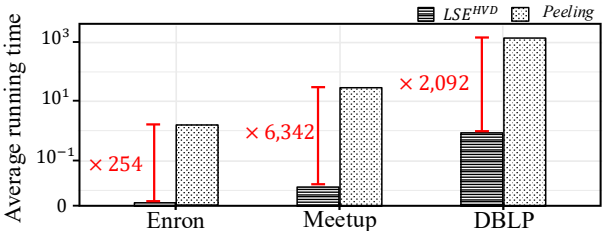


Figure 14: Case study - Size-based query processing

EQ7-2. Case study: Size-based query processing. As a demonstration of applicability, we conduct a case study to explore how our algorithm performs in real-world scenarios. We explore identifying cohesive subgraphs within a specific size range without predetermined k and g using the *Enron*, *Meetup*, and *DBLP* datasets, comparing methods with and without indexing. The LSE^{HVD} indexing technique is used for the index-based method.

To simulate real-world scenarios, size lower bounds are randomly set between 30 and 100, and t, representing the range between the lower and upper bounds, is also randomly chosen between 10 and 100 to generate 10 queries for two methods.

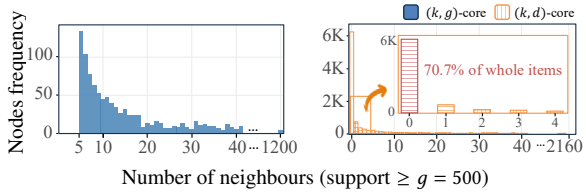
- **Without indexing tree:** The algorithm starts from $g = 1$ and, for each g, incrementally increases k to identify valid (k,g) pairs via the peeling algorithm. If the size of the resulting node set falls below the lower bound for a given g, the algorithm increments g and repeats the search with a fresh (k,g) pair.
- **With indexing tree:** For each increasing value of g, the algorithm utilises the hierarchical structure of the indexing tree to perform two binary searches along the k-side for each fixed g, thereby locating the smallest and largest (k,g)-cores whose sizes satisfy the specified range. All valid cores are collected, and the process continues until no such cores remain.

Figure 14 presents the running times of the two approaches. We observe that the index-based approach is significantly faster than the peeling algorithm. Note that to find all pairs of indices satisfying the size constraint, the algorithm must be executed multiple times. This highlights a core advantage of index-based search: the ability to efficiently explore multiple (k,g)-core candidates under dynamic size constraints. For the Enron dataset, we found an average of 17 pairs, while for the Meetup and DBLP datasets, we found 30 and 36 pairs, respectively. These pairs enable efficient and effective querying within the given size constraints, demonstrating the practical advantages of our index-based approach in real-world applications.

EQ7-3. Case study: Frequent set mining. We evaluate the practicality of the (k,g)-core for frequent set mining by comparing three approaches—Apriori, the (k,d)-core, and the (k,g)-core—on the

Table 5: Summary of frequent set mining

Method	# of items	Avg. neighbour($g \geq 500$)	Avg. degree
Apriori [1]	5	4.0	244,384
(k, g)-core	1,234	47.548	14,865
(k, d)-core [3]	8,746	7.255	3,412



(a) (k, g)**-core** **(b)** (k, d)**-core**

Figure 15: Distribution of neighbors with support $g \geq 500$**.**

Instacart shopping dataset [8], which contains 49,000 items and 3.3M transactions. Using Apriori with a support threshold of $g = 500$, the longest frequent itemset has length 5. We then set $k = 5$ and $d = g = 500$ to compute both cores for comparison.

The results reveal clear subset relationships: all frequent sets identified by Apriori are contained within the (k, g)-core, which itself is fully contained within the (k, d)-core. Table 5 and Figure 15 summarise the outcomes:

- **Apriori:** It produces only small itemsets (maximum length 5), as it requires all items to co-occur in the same transaction. The resulting nodes have very high degrees and strong connectivity, but the coverage remains narrow.
- (k, d)**-core:** It returns 8,746 items. The relaxed constraint admits many items with weak co-occurrence: nodes have an average of only 7.2 neighbours with support ≥ 500, and an average degree of 3,412. Approximately 70.7% of the nodes have no frequent items at the given support threshold, indicating their limited connectivity.
- (k, g)**-core:** It identifies 1,234 items with markedly stronger co-occurrence structure. Nodes average 47.5 neighbours co-occurring at least 500 times, and achieve a substantially higher average degree of 14,865.

Overall, the (k, g)-core provides a balanced trade-off: it identifies a broader set of items than Apriori while maintaining strong co-occurrence, in contrast to the (k, d)-core which admits many weakly connected items.

6 RELATED WORK

6.1 Cohesive Subgraphs Discovery in Hypergraphs

Cohesive subhypergraph discovery has been an active research area, focusing on identifying tightly connected substructures within hypergraphs. Early models, such as the (k, l)-hypercore [30] defined a maximal cohesive subgraph where each node has at least k degree and each hyperedge includes at least l nodes, ensuring node density and edge-richness. To enhance adaptability, the (k, t)-hypercore [9] was introduced, requiring each node to have at least k degree and each hyperedge to contain a certain proportion t of nodes, offering more flexibility in defining cohesive structure. Another significant model, the (k, d)-core [3], focuses on creating a maximal strongly induced subhypergraph, which is a subhypergraph composed of original hyperedges entirely contained within given nodes [5, 13]. In this model, each node has at least k neighbours and is connected to d edges. In contrast, the (k, g)-core model [22] incorporates second-order interactions by requiring each node to have at least k neighbours that co-occur in at least g hyperedges, offering a finer-grained structural perspective. Thus, it allows for a more comprehensive understanding of a core structure within a hypergraph.

6.2 Index-based Cohesive Subgraph Discovery

Index-based approaches for cohesive subgraph discovery have been widely studied in various graph settings, but remain unexplored in hypergraphs. For the (k, p)-core in simple graphs, array-based indices have been designed to support efficient queries [43], while in directed graphs, tabular indices are used for (k, l)-core discovery [15]. In multi-layer graphs, lattice- and tree-based k-core indices have been proposed to enable parallel query processing [33]. For bipartite graphs, the (α, β)-core [14] is effectively managed by the Bi-core index [31], which employs a tree-based structure to maintain results and facilitate queries. These studies demonstrate the utility of indexing for diverse cohesive subgraph models, yet none address the challenges posed by multi-parameter settings in hypergraphs. Our LSE^{H} index shares the idea of single-parameter compression with the Bi-core index but introduces key innovations tailored to hypergraphs: unlike the Bi-core index, which stores all nodes of each core in a single array, our framework employs leaf nodes for more flexible storage management. Furthermore, we extend beyond single-parameter compression by proposing two additional schemes: the LSE^{HV} index, which captures multi-dimensional hierarchical relationships, and LSE^{HVD}, which incorporates relationships across non-hierarchical cores.

7 CONCLUSION

This work presents efficient indexing structures for (k, g)-core decomposition in hypergraphs, supporting scalable and adaptive query processing. Recognising the significant challenge of selecting appropriate user parameters, our method enables users to dynamically adjust the values of k and g through the constructed indexing structure. We introduce two indexing approaches: the Naïve indexing approach and the Locality based Space Efficient indexing approach. Additionally, we propose three indexing techniques aimed at mitigating space complexity. Extensive experiments on both real and synthetic hypergraphs validate the effectiveness of the proposed indexing structures in terms of space usage and query latency. Future work includes extending our framework to support dynamic hypergraphs by developing maintenance algorithms that incrementally update the index in response to hypergraph modifications, thereby enabling adaptive and real-time query support in evolving settings.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. RS-2025-00523578, No. RS-2023-00278009, No. RS-2023-00277907), and by the Institute of Information & Communications Technology Planning & Evaluation (IITP) under the Information Technology Research Center (ITRC) program funded by the Korea government (MSIT) (IITP-2025-RS-2021-II211817).

Artifacts

https://github.com/Song1940/kg_decomposition_index

References

[1] Rakesh Agrawal and Ramakrishnan Srikant. 1994. Fast Algorithms for Mining Association Rules in Large Databases. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB '94)*. Morgan Kaufmann, San Francisco, CA, United States, 487–499.

[2] Esra Akbas and Peixiang Zhao. 2017. Truss-based community search: a truss-equivalence based indexing approach. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1298–1309.

[3] Naheed Anjum Arafat, Arijit Khan, Arpit Rai, and Bishwamittra Ghosh. 2023. Neighborhood-Based Hypergraph Core Decomposition. *Proceedings of the VLDB Endowment* 16 (07 2023), 2061–2074. doi:10.14778/3598581.3598582

[4] Joonhyun Bae and Sangwook Kim. 2014. Identifying and ranking influential spreaders in complex networks by neighborhood coreness. *Physica A: Statistical Mechanics and its Applications* 395 (2014), 549–559.

[5] Mohammad A Bahmanian and Mateja Sajna. 2015. Connection and separation in hypergraphs. *Theory and Applications of Graphs* 2, 2 (2015), 5.

[6] Vladimir Batagelj and Matjaz Zaversnik. 2003. An o (m) algorithm for cores decomposition of networks. *arXiv preprint cs/0310049* (2003).

[7] Austin R Benson, Rediet Abebe, Michael T Schaub, Ali Jadbabaie, and Jon Kleinberg. 2018. Simplicial closure and higher-order link prediction. *Proceedings of the National Academy of Sciences* 115, 48 (2018), E11221–E11230.

[8] Austin R Benson, Ravi Kumar, and Andrew Tomkins. 2018. A discrete choice model for subset selection. In *Proceedings of the ACM International Conference on Web Search and Data Mining*. Association for Computing Machinery, New York, NY, United States, 37–45.

[9] Fanchen Bu, Geon Lee, and Kijung Shin. 2023. Hypercore decomposition for non-fragile hyperedges: concepts, algorithms, observations, and applications. *Data Mining and Knowledge Discovery* 37, 6 (2023), 2389–2437.

[10] Evo Busseniers. 2014. General Centrality in a hypergraph. *arXiv preprint arXiv:1403.5162* (2014).

[11] Philip S Chodrow, Nate Veldt, and Austin R Benson. 2021. Hypergraph clustering: from blockmodels to modularity. *Science Advances* (2021).

[12] Deming Chu, Fan Zhang, Xuemin Lin, Wenjie Zhang, Ying Zhang, Yinglong Xia, and Chenyi Zhang. 2020. Finding the best k in core decomposition: A time and space optimal solution. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, IEEE, 685–696.

[13] Megan Dewar, John Healy, Xavier Pérez-Giménez, Pawel Pralat, John Proos, Benjamin Reiniger, and Kirill Ternovsky. 2017. Subhypergraphs in non-uniform random hypergraphs. *Internet Mathematics* (03 2017). doi:10.24166/im.03.2018

[14] Danhao Ding, Hui Li, Zhipeng Huang, and Nikos Mamoulis. 2017. Efficient fault-tolerant group recommendation using alpha-beta-core. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. 2047–2050.

[15] Yixiang Fang, Zhongran Wang, Reynold Cheng, Hongzhi Wang, and Jiafeng Hu. 2018. Effective and efficient community search over large directed graphs. *IEEE Transactions on Knowledge and Data Engineering* 31, 11 (2018), 2093–2107.

[16] Song Feng, Emily Heath, Brett Jefferson, Cliff Joslyn, Henry Kvinge, Hugh D Mitchell, Brenda Praggastis, Amie J Eisfeld, Amy C Sims, Larissa B Thackray, et al. 2021. Hypergraph models of biological networks to identify genes critical to pathogenic viral response. *BMC bioinformatics* 22, 1 (2021), 1–21.

[17] Edoardo Galimberti, Francesco Bonchi, Francesco Gullo, and Tommaso Lanciano. 2020. Core decomposition in multilayer networks: Theory, algorithms, and applications. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 14, 1 (2020), 1–40.

[18] Hector Garcia-Molina. 2008. *Database systems: the complete book*. Pearson Education India.

[19] Sherry He, Brett Hollenbeck, Gijs Overgoor, Davide Proserpio, and Ali Tosyali. 2022. Detecting fake-review buyers using network structure: Direct evidence from Amazon. *Proceedings of the National Academy of Sciences* 119, 47 (2022), e2211932119.

[20] DJ-H Huang and Andrew B Kahng. 1995. When clusters meet partitions: new density-based methods for circuit decomposition. In *Proceedings the European Design and Test Conference. ED&TC 1995*. IEEE, 60–64.

[21] Jin Huang, Rui Zhang, and Jeffrey Xu Yu. 2015. Scalable hypergraph learning and processing. In *2015 IEEE International Conference on Data Mining*. IEEE, 775–780.

[22] Dahee Kim, Junghoon Kim, Sungsu Lim, and Hyun Ji Jeong. 2023. Exploring Cohesive Subgraphs in Hypergraphs: The (k, g)-core Approach. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management*. 4013–4017.

[23] Junghoon Kim, Tao Guo, Kaiyu Feng, Gao Cong, Arijit Khan, and Farhana M Choudhury. 2020. Densely connected user community and location cluster search in location-based social networks. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 2199–2209.

[24] Song Kim, Dahee Kim, Taejoon Han, Junghoon Kim, Hyun Ji Jeong, and Jungeun Kim. Online appendix of Efficient Locality-based Indexing for Cohesive Subgraphs Discovery in Hypergraphs. Available at: https://github.com/Song1940/kg_decomposition_index/blob/main/Appendix.pdf.

[25] Maksim Kitsak, Lazaros K Gallos, Shlomo Havlin, Fredrik Liljeros, Lev Muchnik, H Eugene Stanley, and Hernán A Makse. 2010. Identification of influential spreaders in complex networks. *Nature physics* 6, 11 (2010), 888–893.

[26] Jihoon Ko, Yunbum Kook, and Kijung Shin. 2022. Growth patterns and models of real-world hypergraphs. *Knowledge and Information Systems* 64, 11 (2022), 2883–2920.

[27] Tarun Kumar, K Darwin, Srinivasan Parthasarathy, and Balaraman Ravindran. 2020. HPRA: Hyperedge prediction using resource allocation. In *Proceedings of the 12th ACM conference on web science*. 135–143.

[28] Fangyuan Lei, Jiahao Huang, Jianjian Jiang, Da Huang, Zhengming Li, and Chang-Dong Wang. 2024. Unveiling the potential of long-range dependence with mask-guided structure learning for hypergraph. *Knowledge-Based Systems* 284 (2024), 111254.

[29] Ming Leng, Lingyu Sun, Ji-nian Bian, and Yuchun Ma. 2013. An o(m) algorithm for cores decomposition of undirected hypergraph. *Journal of Chinese Computer Systems* 34, 11 (2013), 2568–2573.

[30] Stratis Limnios, George Dasoulas, Dimitrios M Thilikos, and Michalis Vazirgiannis. 2021. Hcore-init: Neural network initialization based on graph degeneracy. In *2020 25th International Conference on Pattern Recognition (ICPR)*. IEEE, 5852–5858.

[31] Boge Liu, Long Yuan, Xuemin Lin, Lu Qin, Wenjie Zhang, and Jingren Zhou. 2019. Efficient (α, β)-core computation: An index-based approach. In *The World Wide Web Conference*. 1130–1141.

[32] Boge Liu, Fan Zhang, Wenjie Zhang, Xuemin Lin, and Ying Zhang. 2021. Efficient community search with size constraint. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 97–108.

[33] Dandan Liu, Run-An Wang, Zhaonian Zou, and Xin Huang. 2024. Fast Multilayer Core Decomposition and Indexing. In *2024 IEEE 40rd international conference on data engineering (ICDE)*. IEEE, 2695–2708.

[34] Linyuan Lü, Tao Zhou, Qian-Ming Zhang, and H Eugene Stanley. 2016. The H-index of a network node and its relation to degree and coreness. *Nature communications* 7, 1 (2016), 10168.

[35] Rodica Ioana Lung, Noémi Gaskó, and Mihai Alexandru Suciu. 2018. A hypergraph model for representing scientific output. *Scientometrics* 117 (2018), 1361–1379.

[36] Shashank Pandit, Duen Horng Chau, Samuel Wang, and Christos Faloutsos. 2007. Netprobe: a fast and scalable system for fraud detection in online auction networks. In *Proceedings of the 16th international conference on World Wide Web*. 201–210.

[37] 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*. 939–948.

[38] Huanhuan Wu, James Cheng, Yi Lu, Yiping Ke, Yuzhen Huang, Da Yan, and Hejun Wu. 2015. Core decomposition in large temporal graphs. In *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, 649–658.

[39] Xin Xia, Hongzhi Yin, Junliang Yu, Qinyong Wang, Lizhen Cui, and Xiangliang Zhang. 2021. Self-supervised hypergraph convolutional networks for session-based recommendation. In *Proceedings of the AAAI conference on artificial intelligence*. 4503–4511.

[40] Bowen Yan and Jianxi Luo. 2019. Multicores-periphery structure in networks. *Network Science* 7, 1 (2019), 70–87.

[41] Kai Yao and Lijun Chang. 2021. Efficient size-bounded community search over large networks. *Proceedings of the VLDB Endowment* 14, 8 (2021), 1441–1453.

[42] Zhouxin Yu, Jintang Li, Liang Chen, and Zibin Zheng. 2022. Unifying multi-associations through hypergraph for bundle recommendation. *Knowledge-Based Systems* 255 (2022), 109755.

[43] Chen Zhang, Fan Zhang, Wenjie Zhang, Boge Liu, Ying Zhang, Lu Qin, and Xuemin Lin. 2020. Exploring finer granularity within the cores: Efficient (k, p)-core computation. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 181–192.

[44] Fan Zhang, Qingyuan Linghu, Jiadong Xie, Kai Wang, Xuemin Lin, and Wenjie Zhang. 2023. Quantifying Node Importance over Network Structural Stability. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 3217–3228.