Appendix A NEW PEELING ALGORITHM FOR (k, g)-CORE

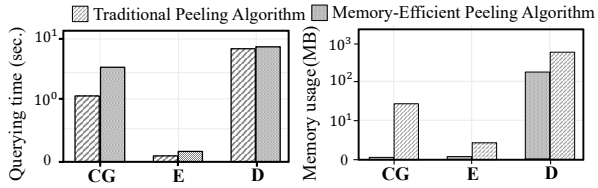
Algorithm 1 presents the overall procedure of (k, g)-core computation.

Algorithm 1: Memory-Efficient Peeling Algorithm

Input: Hypergraph $G = (V, E)$, parameters k and g
Output: The (k, g)-core of G

```
changed ← true, H ← V;
while changed do
    changed ← false;
    nodes ← H;
    foreach v ∈ nodes do
        N(v) ← {(w, cnt(v, w)) | w ∈ V, and cnt(v, w) ≥ g};
        if |N(v)| < k then
            changed ← true;
            H ← H \ {v};
return H
```

As discussed in Section 2, we develop a new memory-efficient (k, g)-core algorithm that serves as the foundation for our index construction methods. Compared to the previous (k, g)-core peeling algorithm in [3], which requires $O(|V|^2)$ space, our memory-efficient algorithm reduces space usage to $O(|V|)$ by iteratively computing co-occurrence counts. However, this comes at the cost of increased time complexity, rising to $O(|V| \cdot B \cdot |e^*|)$ from $O(B \cdot |e^*|)$ where $B = \sum_{e \in E} |e|$ and $|e^*| = \max_{e \in E} |e|$, due to repeated neighbour computations. Despite this higher time complexity, our experiments show that the actual running time is not significantly different. Figure 1 compares the performance of both algorithms across various real-world datasets, evaluating both querying time and memory efficiency. Using the same settings as in EQ2 to select appropriate k and g values, we measured the average querying time. As shown in Figure1a, the querying times of both algorithms are nearly identical. In terms of memory efficiency, our algorithm consistently uses less memory. These results demonstrate the effectiveness and practicality of our new algorithm for large-scale hypergraphs.



(a) Querying time comparison **(b) Memory usage comparison**

Figure 1: Comparison with peeling algorithm [3]

Appendix B ADDITIONAL EXPERIMENTS

Appendix B.1 Leaf node size distribution

To gain insights into the locality among the (k, g)-cores and to understand the impact of each indexing technique, we analyse the size distribution of leaf nodes in the indexing trees. Figure 2 presents the leaf node size distribution for each indexing tree in the *Contact* dataset. In the Naïve indexing tree, we observe that leaf nodes with relatively lower g values tend to have larger sizes. This suggests a concentration of nodes in these lower-order cores. However, in the case of the LSE^H indexing tree, there is a notable reduction in the total number of nodes across all leaf nodes, indicating that a substantial amount of duplication has been eliminated. Despite this reduction, a significant number of large-sized leaf nodes still exist, especially in the central part of the indexing tree. Further processing of the LSE^{HV} and LSE^{HVD} indexing techniques leads to an even more substantial reduction in duplicate nodes. As a result, only a few leaf nodes, particularly those with higher k and g values, exhibit relatively large sizes. This trend indicates a clear pattern: as the additional indexing progresses, nodes tend to converge towards leaf nodes with higher k and g values. This convergence is indicative of the hierarchical nature of (k, g)-cores and points out the efficiency of our indexing techniques in organising and revealing this structure.

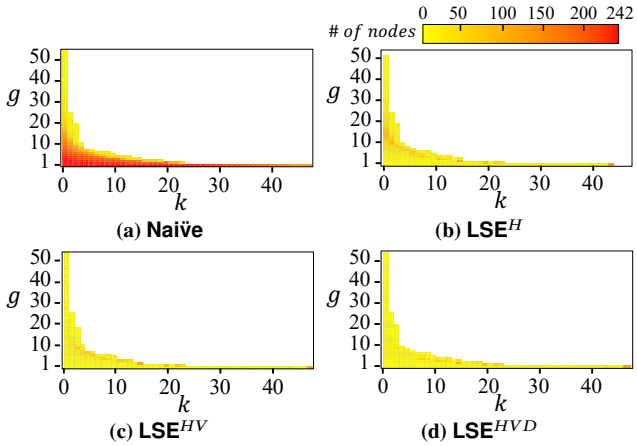
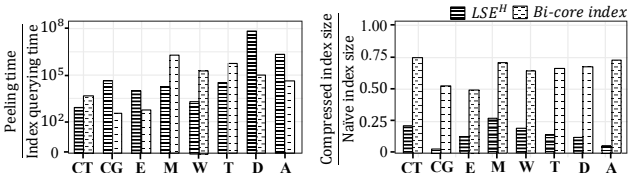


Figure 2: Distributions of nodes for each indexing tree



(a) Relative querying time **(b) Relative node size**

Figure 3: Comparison with Bi-core index

Appendix B.2 Comparision with Bi-core index

To compare our approach with other indexing-based models, we evaluate it against the (α, β)-core index model. First, we evaluate query performance relative to the peeling-based approach. The Bi-core index [5] is designed to identify (α, β)-cores (bi-core) in bipartite graphs, where each node on the left and right sides has degree at least α and β, respectively. We measured the speedup of the Bi-core index querying algorithm over the bi-core peeling algorithm proposed in [2]. We then compare the Bi-core index with our LSE^H indexing tree, as both Bi-core and LSE^H indexing methods fix one

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	○

parameter and iterate over the other to compute cores, resulting in comparable structural characteristics. Specifically, we computed the speed of LSE^H indexing tree querying relative to the (k,g)-core peeling algorithm described in Section 2. Query selection follows the method outlined in **EQ2**. Figure 3a shows the execution time ratios between index-based querying and their corresponding peeling algorithms for both approaches. Our results indicate that LSE^H indexing tree querying achieves performance comparable to, or even better than, the Bi-core index in most datasets, despite handling a more complex neighbourhood structure. This demonstrates that our method remains efficient even under increased structural complexity.

Figure 3b compares the node size ratios between the Naïve and compressed versions of the Bi-core index and our indexing method (Naïve indexing tree compared to LSE^H indexing tree). Although both indices operate in a similar manner, the Bi-core index retains more than half, and up to 75%, of its original nodes even after compression. In contrast, our LSE^H indexing tree retains less than 25%, and in some cases less than 5%, of the nodes in the Naïve indexing tree. These results demonstrate the strong locality captured by the (k,g)-core model, indicating that many nodes are shared across cores with similar coreness values. This structural overlap justifies our indexing-based decomposition strategy and explains its efficiency in compressing redundant information. Such effectiveness is especially relevant for hypergraphs, which naturally model higher-order inter-cohesion among groups of nodes, often resulting in substantial overlap between dense regions.

Appendix C SUMMARY OF ALGORIOTHMS

Summary of Algorithms. Each indexing technique is designed to minimise memory usage while ensuring efficient query processing. Table 1 provides an efficiency analysis of the algorithms, including their time and space complexities. The table also details the leaf node structures, types of links, and the presence of auxiliary nodes for each resulting index from algorithm. We use a star notation ($\star$) to indicate efficiency levels, where a higher number of stars represents greater efficiency. The table highlights a clear trade-off between query processing efficiency and space efficiency. Note that even with identical complexity, the actual performance can vary significantly, as complexity analysis considers extreme scenarios that may not fully reflect the 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 [1, 4, 6]. 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 [7, 8]. This demonstrates that our indexing structures are not only memory-efficient but also support meaningful structural interpretation of hypergraph topology.

REFERENCES

[1] J. Bae and S. Kim. Identifying and ranking influential spreaders in complex networks by neighborhood coreness. *Physica A: Statistical Mechanics and its Applications*, 395:549–559, 2014.

[2] D. Ding, H. Li, Z. Huang, and N. Mamoulis. Efficient fault-tolerant group recommendation using alpha-beta-core. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pages 2047–2050, 2017.

[3] D. Kim, J. Kim, S. Lim, and H. J. Jeong. Exploring cohesive subgraphs in hypergraphs: The (k,g)-core approach. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management*, pages 4013–4017, 2023.

[4] M. Kitsak, L. K. Gallos, S. Havlin, F. Liljeros, L. Muchnik, H. E. Stanley, and H. A. Makse. Identification of influential spreaders in complex networks. *Nature physics*, 6(11):888–893, 2010.

[5] B. Liu, L. Yuan, X. Lin, L. Qin, W. Zhang, and J. Zhou. Efficient (α, β)-core computation: An index-based approach. In *The World Wide Web Conference*, pages 1130–1141, 2019.

[6] L. Lü, T. Zhou, Q.-M. Zhang, and H. E. Stanley. The h-index of a network node and its relation to degree and coreness. *Nature communications*, 7(1):10168, 2016.

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

[8] F. Zhang, Q. Linghu, J. Xie, K. Wang, X. Lin, and W. Zhang. Quantifying node importance over network structural stability. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 3217–3228, 2023.