Supplementary Material – Fast Multilayer Core Decomposition and Indexing

A. Supplementary Explanation to MLCS (Algorithm 4).

We here present a more detailed explanation to the MLC-tree-based ML core search algorithm MLCS in Algorithm 4.

As stated in Section IV-B1, when provided with a vector $\mathbf{k}$, MLCS first calls Procedure SEARCH to locate the $\mathbf{k}$-node N in the MLC-tree and then invokes Procedure RECOVER to restore the ML core represented by N.

Procedure SEARCH (lines 3–8) works as follows: it repeatedly calls Procedure FORWARD to find the path from the root node of the MLC-tree to the $\mathbf{k}$-node. During this process, we maintain a node N, along with a counter i, initialized to the root of the MLC-tree and 1, respectively. Each call to Procedure FORWARD examines the coreness vector represented by N, denoted as $\mathbf{k}_N$. If $\mathbf{k}_N[i] < \mathbf{k}[i]$, the procedure forwards to search the child node N' of N with vectors different from $\mathbf{k}_N$ in their i-th components and thereby returns N' with i (lines 10–13). Otherwise, the i-th component of $\mathbf{k}_N$ already matches that of $\mathbf{k}$. It then returns N and $i+1$ to move on to the matching of their $(i+1)$-th components (lines 14–15). Procedure FORWARD establishes the following invariant, which ensures the correctness of Procedure SEARCH:

Lemma 1. *Given N and i as input, let N' and i' be the output of Procedure* FORWARD. *N' must be on the path from the root to the $\mathbf{k}$-node in the MLC-tree. And if $i' = i+1$, we have that $\mathbf{k}_{N'}[j] = \mathbf{k}[j]$ for $j = 1, 2, \ldots, i$.*

Procedure RECOVER (lines 16–23) repeatedly collects the vertex sets associated with the nodes along the rightmost path starting from N down to a leaf node and returns the union of these vertex sets as the result. This process correctly restores the ML core represented by N, as guaranteed by Theorem 3.

B. Supplements for the MLC-tree-based WDS Algorithm.

The pseudocode for the MLC-tree-based WDS algorithm is given in Algorithm 5. It makes a DFS traversal on the MLC-tree and extracts the ML core maximizing the weighted density as the result. The basic DFS traversal framework is clear from the pseudocode, and we detail the search process on a given rightmost path, which is outlined in Procedure SEARCHRMPATH (lines 12–25).

Procedure SEARCHRMPATH takes an MLC-tree T and a node N as arguments. It returns the node N_{dense} representing the ML core with the highest weighted density on N's rightmost path with its density ρ, the number n of vertices in the ML core represented by N, and the per-layer edges numbers in the subgraph induced by this ML core, which is stored in an array m. The procedure starts by going straight to the leaf node of N's rightmost path (lines 13–15) and then identifies the densest ML core on the path during backtracking. Before initiating backtracking, n and all components of m are set to 0 (line 17). When backtracking to node N, letting C be the ML core represented by the rightmost child of N, n and m already store the size of C and the per-layer edges in the subgraph induced by C, respectively (line 15). We then obtain the vertex set S associated with N in line 18. Theorem 3 guarantees that the ML core represented by N equals $C \cup S$. Therefore, the number of vertices in it can be restored by adding $|S|$ to n (line 19). However, restoring the per-layer edge numbers is a bit tricky (lines 20–21): it has to call Procedure GetDiffEdge to compute the number of newly introduced edges ΔE_i due to the involvement of S in each layer i, i.e., the ones with one endpoint in S while the other in $S \cup C$, and then add ΔE_i to $m[i]$. Next, the weighted density of the ML core represented by N can be computed using Procedure GetWeightedDensity in line 22 based on n and m. If this ML core attains a higher density than any other ML cores on N's rightmost path, N_{dense}, and ρ are accordingly updated (lines 23–24).

Algorithm 5 MLWDS (Weighted densest subgraph search)

Input: The MLC-tree T for an ML graph $\mathcal{G} = (V, E, L)$, a positive real number β, and $|L|$ positive real weights $w_1, w_2, \cdots, w_{|L|}$
Output: The vertex set C with approximate the highest weighted density (Eq. (1)), with its density ρ

```
1: N ← the root of the MLC-tree T
2: N_dense, ρ ← SEARCHWDS(T, N)
3: C ← RECOVER(T, N_dense)
4: return C, ρ
5: procedure SEARCHWDS(T, N)
6:     N_dense, ρ ← SEARCHRMPATH(T, N)
7:     for each non-rightmost child N' of N do
8:         N'_dense, ρ' ← SEARCHWDS(T, N')
9:         if ρ' > ρ then
10:            ρ ← ρ', N_dense ← N'_dense
11:    return N_dense, ρ
12: procedure SEARCHRMPATH(T, N)
13:    if N is not a leaf node then
14:        N' ← N's rightmost child
15:        N_dense, ρ, n, m ← SEARCHRMPATH(T, N')
16:    else
17:        ρ ← 0, n ← 0, m ← {0, 0, ··· , 0}
18:    S ← the vertex set associated with N
19:    n ← n + |S|
20:    for i ← 1, 2, ··· , |L| do
21:        m[i] ← m[i] + GETDIFFEDGE(S)
22:    ρ' ← GETWEIGHTEDDENSITY(n, m)
23:    if ρ' > ρ then
24:        ρ ← ρ', N_dense ← N
25:    return N_dense, ρ, n, m
```

C. Supplements to Experimental Settings.

We here supplement some experimental settings not mentioned in the paper for reproducibility.

1) All ML core decomposition algorithms proposed in this paper are executed on ML graphs with layers sorted in a non-decreasing order of layer density.
2) The hash table used for storing the ML core decomposition is implemented with separate chaining, where the number of buckets is set to 2^b, with $b = \lceil \log_2 n \rceil$ and n representing the number of nonempty ML cores in the respective graph. The hashing function employed is `hash_range` from the Boost library [1].

D. Impact of Layer Orders.

TABLE III
RUNNING TIME (S) OF MLCD W.R.T. LAYER ORDERS

Graph	Random	Density/Degeneracy↑	Density/Degeneracy↓
SC	16.25	**7.18**	118.32
DS	**4.44**	4.87/5.11	4.68/4.66
OI	5.67	**3.65**	5.65
A	76.02	**75.23**	130.21
H	237.66	**10.36**	240.38
Ff	44.70	**44.65**	184.85
DL	1367.25	**391.70**/434.52	10638.20/10013.60
W	354.26	333.98/**165.47**	4512.40/4570.11
FG	–	**31355.30**/38318.90	–

TABLE IV
MEMORY USAGE (MB) OF THE MLC-TREE W.R.T. LAYER ORDERS

Graph	Random	Density/Degeneracy↑	Density/Degeneracy↓
SC	28.15	**28.03**	225.61
DS	10.02	9.98/**9.95**	9.97/10.01
OI	35.32	**25.12**	35.32
A	**198.84**	**198.84**	371.86
H	255.42	**14.19**	255.42
Ff	**80.36**	**80.36**	197.41
DL	622.77	617.26/**616.05**	1493.33/1496.66
W	225.62	225.62/**121.80**	681.86/681.57
FG	–	**19436.90**	–

In this experiment, we investigate the impact of different layer orders on the efficiency of ML core decomposition and the memory cost of the constructed MLC-tree index. We tested a total of five orders, including random order (Random), non-decreasing order of layer density or degeneracy (Density/Degeneracy↑), and non-increasing order of layer density or degeneracy (Density/Degeneracy↓). As sorting layers by density and degeneracy usually yields the same order, we report a single value for the running time or memory cost in the table; if the resulting orders differ, two values are shown.

The results on the decomposition efficiency are reported in Table III, which align with our expectations: ordering layers in non-decreasing order of their density or degeneracy generally incurs minimal time overhead for most graphs. It is also notable that layer orders influence the decomposition efficiency in ML graphs with wide layer density/degeneracy ranges more significantly. For example, in graph Higgs (H), where the densest layer has a density $445\times$ larger and a degeneracy $41\times$ larger than the sparsest layer, sorting layers in the non-decreasing order of layer density leads to a substantial speedup ratio of $23.2\times$ compared to sorting in the non-increasing order.

The results on memory costs of the MLC-tree index are presented in Table IV. In line with the analysis in Section IV-A, the non-decreasing order of layer degeneracy always yields the smallest MLC-tree. Sorting layers in the non-decreasing order of the layer degeneracy is observed to show varying levels of space reductions on graphs DBLP-Large (DL) and Wiki (W) compared with sorting based on the layer density. On graph DL, these two orders guarantee the same $|L|$-th layer, leading to identical costs for storing differences along the rightmost paths. However, sorting by degeneracy results in fewer nodes with small $lnz(\mathbf{k})$ values, contributing to smaller space costs for storing the tree nodes. In contrast, the significant space reduction observed in graph FG is attributed to the decreased number of rightmost paths that need to be stored.

E. Discussions on MLC-tree Maintenance

The dynamic nature of real-world graphs necessitates effective strategies for maintaining the MLC-tree index in the face of frequent graph alterations. However, this task is notably more challenging compared to maintaining indexes of cohesive subgraphs proposed for single-layer graphs due to the following two aspects:

1) Multi-layer graphs introduce additional dimensions of graph changes beyond traditional ones such as adding or removing nodes and edges, namely adding or removing layers, which complicates the update process.
2) Changes in one layer can have severe cascading effects on other layers through the correspondence of cross-layer nodes.

We here provide an overview of our approach to addressing this issue and leave the technical details to future work. Given that the insertion or deletion of a vertex can equivalently be viewed as the insertion or deletion of all its incident edges, we focus on updating the MLC-tree in response to layer insertion/deletion and edge insertion/deletion events.

1) Layer Update: We consider the cases for layer insertion and deletion separately.

Adding a new layer. When a new layer is added to an ML graph $\mathcal{G} = (V, E, L)$, we designate this layer as G_0 for simplicity, with the smallest ordering. Subsequently, each original layer G_l in $\mathcal{G}$ becomes G_{l+1}. Following the addition, the original MLC-tree can be viewed as a subtree within the new tree, representing ML cores with no constraints from G_0, i.e., $\mathbf{k}[0] = 0$. Therefore, we refine the vector associated with each original $\mathbf{k}$-node by inserting a 0 before its first component, e.g., the $(0, 1)$-node will become the $(0, 0, 1)$-node. To incorporate nodes representing ML cores with constraints from G_0 while maintaining the required tree structure for ML core searching, we introduce a new child, denoted as N, to the $\mathbf{0}^{|L|+1}$-node (the root of the MLC-tree). This new child's vector is generated by incrementing the first component of

$\mathbf{0}^{|L|+1}$ by 1. Subsequently, we apply Algorithm 1 or its parallel versions to compute the subtree rooted at N.

Removing a layer. Suppose G_l, where $1 \leq l \leq |L|$, is removed, it can be seen as all constraints imposed on G_l from $\mathbf{k}[l]$ are removed. There are two cases:

a) If $l < |L|$, all $\mathbf{k}$-nodes with $\mathbf{k}[l] > 0$ can be removed from the MLC -tree. Specifically, for each $\mathbf{k}$-node with $\mathbf{k}[l] = 0$ in the MLC-tree, we remove the entire subtree rooted at its child $\mathbf{k}'$-node, where $\mathbf{k}$ and $\mathbf{k}'$ differ at the l-th component. Then, for all remaining $\mathbf{k}$-node, we refine its vector by removing the l-th component. For example, the $(0, 1, 0)$-node becomes the $(1, 0)$-node if $l = 0$.

b) If $i = |L|$, we adopt the procedure in step a) to remove the nodes with $\mathbf{k}[l] > 0$. However, in this case, all rightmost paths of the original MLC-tree and their associated vertex sets are removed, and therefore, we have to reconstruct the index structure by recomputing the vertex subsets associated with nodes on each new rightmost path. Nevertheless, we are able to realize this in a more efficient way: for each pair of father-child nodes on new rightmost paths, we search the ML cores represented by them from the original MLC-tree, compute the difference between them, and associate it with the father node.

2) Edge Update: We primarily focus on edge insertion as a representative. The solution for handling the deletion case can largely be derived from the insertion case and will be explored in future work.

Without loss of generality, let us suppose an edge $e = (u, v, l)$ is inserted in G_l, where $1 \leq l \leq |L|$. To update the MLC-tree index against an insertion, we first need to identify the affected area of the insertion, i.e., which ML cores are changed. We decompose this problem into a series of subproblems. Specifically, let $L' = L - \{l\}$. Given $\mathbf{k}_{L'}$, a $|L|$-dimensional vector with $\mathbf{k}_{L'}[l] = 0$, describing the cohesiveness constraints imposed on layers in L', we denote $\mathsf{cn}_{\mathbf{k}_{L'}}(u)$ as the maximum integer k such that u is in a nonempty $\mathbf{k}$-core, where $\mathbf{k}[l] = k$ and $\mathbf{k}[l'] = \mathbf{k}_{L'}[l']$ for $1 \leq l' \leq |L|$ and $l' \neq l$. It's easy to determine that the ML $\mathbf{k}$-core comprises all vertices u such that $\mathsf{cn}_{\mathbf{k}_{L'}}(u) \geq \mathbf{k}[l]$. Following the single-layer counterpart [2], we can deduce that only the ML $\mathbf{k}$-cores with $\mathbf{k}[l] \leq \min\{\mathsf{cn}_{\mathbf{k}_{L'}}(u), \mathsf{cn}_{\mathbf{k}_{L'}}(v)\}+1$ if $\mathsf{cn}_{\mathbf{k}_{L'}}(u) = \mathsf{cn}_{\mathbf{k}_{L'}}(v)$, or $\mathbf{k}[l] \leq \min\{\mathsf{cn}_{\mathbf{k}_{L'}}(u), \mathsf{cn}_{\mathbf{k}_{L'}}(v)\}$ otherwise, may change. Therefore, we update the necessary vertex subsets attached to the MLC-tree related to these ML cores. This process is repeated for every possible value of $\mathbf{k}'_L$, completing the update for the MLC-tree.

It's certain that the identified affected areas mentioned above may be larger than the actual ones. Our future efforts will focus on refining this estimation to minimize unnecessary updates. Additionally, we will delve into studying effective strategies for updating the vertex subsets.

Due to the extensive impact of a single-edge change, it's necessary to investigate the maintenance method for batch edge updates, in which we can exploit the nullifying effects caused by different edge changes and the possibilities for aggregate updates. In addition, it is worth studying the scenarios where updating the MLC-tree might lag behind recomputing the entire tree.

F. Making the MLC-tree Index Smaller

Improving the space efficiency of the MLC-tree index plays a crucial role in enhancing its practicality and applicability, particularly in situations with limited memory or when dealing with large graphs. We outline below three potential directions to reduce the index space overhead:

1) Using the *on-the-fly tree compaction technique* that performs the detection and reusing of the redundant subtrees in the MLC-tree during the construction. With the premise of ensuring that Algorithm 4 still functions after performing the reduction, we have explored the possibility of leveraging the P-tree compaction techniques [3], originally proposed for reducing redundancy in the KP-tree index for gCores, to address redundancy in the MLC-tree. This approach naturally fits in this case as the key features (uniqueness and containment relationships) of gCores and ML cores are similar. Specifically, we incorporate in the MLC-tree construction the detection of structural-isomorphic subtrees sharing vertex subsets and removing redundant nodes/subtrees or merging them with other isomorphic ones. This approach not only reduces the space costs of the index but also speeds up the construction process. However, implementing this on-the-fly compaction approach in parallel poses challenges as it heavily relies on the generation order of nodes. To make this compaction approach more practical, we are committed to further investigating the possibility of employing compaction in parallel settings as part of our future work.

2) Using *engineering-focused approaches* or data compression methods. Reducing redundancy in the vertex subsets associated with the MLC-tree nodes contributes to smaller index space costs. A straightforward approach would be using a hash set to store each unique vertex subset and associate these subsets to the respective nodes via pointers. Additionally, leveraging data compression methods that exploit the overlap between vertex subsets, such as the rule-based text-data compression method [4], [5], can further enhance space efficiency.

3) *Eliminating tree nodes* with guaranteed precision on the ML core search results. We can significantly reduce the space costs of the index at the expense of slightly longer search time or lower qualified results. Specifically, we can generate the child nodes by increasing the components of the father's vector by a fixed step greater than 1 or by a variable step such as doubling the components. If relaxing the result accuracy is allowed, we can slightly modify Algorithm 4 to search for the descendant core or ancestor core of the queried one as the result. Otherwise, a refinement from the ancestor core to the queried core is necessary. Furthermore, considering that not all ML cores are of interest to users, especially those with small $\mathbf{k}$ values, it's feasible to maintain partial ML cores that align with users' interests. It can be realized by adopting heuristic or learning methods based on historical queries.

G. Discussions on the Parallel Paradigm

The parallel paradigm introduced in Section III(c), along with the accompanying optimizations, can be simply adapted to solve the gCore decomposition problem [3] for general ML graphs. In general ML graphs, the vertex sets across layers may vary, and the connections between vertices across layers can exhibit arbitrary patterns. The notion of gCore tailored for such graphs is characterized by two parameters: $\mathbf{k} \in \mathbb{N}^{|L|}$ and $\mathbf{p} \in [0,1]^{|L|-1}$, and the gCore is unique given vectors $\mathbf{k}$ and $\mathbf{p}$. To efficiently address the gCore decomposition problem, the authors propose organizing all potential gCores into a tree-of-trees structure termed KP-tree. The outermost tree organizes all $\mathbf{k}$ values, using the same layout as the MLC-tree. Differently, each $\mathbf{k}$-node is associated with another tree called P-tree, which organizes all possible values for $\mathbf{p}$ also in a manner akin to the organization of the MLC-tree. This hierarchical KP-tree design guarantees that each node in the P-tree inside the $\mathbf{k}$-node uniquely corresponds to a gCore, and the gCore represented by a father node is always a superset of the one represented by its children.

Clearly, we can employ our rightmost-path-decomposition to each P-tree, and designate available threads to independently compute gCores represented by each path. The path merging strategy can also be directly integrated into this process to enhance efficiency. Employing the core-level-parallel framework in the startup phase can be realized by adapting Procedure `P-Peel` (Algorithm 3) for computing gCores. However, gCores are typically smaller, and the interconnections between layers may cause more contention when updating a vertex's degrees or state. Therefore, a dedicated parallel implementation is required to mitigate these challenges effectively.

Furthermore, we have the opportunity to interleave the computation across multiple P-trees. As the generation of some P-tree nears completion, idle threads can seamlessly transition to processing other P-trees, thereby increasing the overall computational efficiency.

H. Missing Proofs.

1) Supplementary explanations to the proof of Theorem 1: In the proof of Theorem 1, the time for maintaining the result set R in line 10 of Algorithm 1 is $O(\prod_{i=1}^{|L|} \kappa(G_i)|V|)$, which is not included in the analysis. In fact, if we replace line 10 with materializing the $\mathbf{k}$-node and associating the set of vertices removed by PEEL (line 8) to the father of the $\mathbf{k}$-node, an MLC-tree index (Section IV-A) will be built after the algorithm terminates. This process incurs a time cost of $O(\prod_{i=1}^{|L|-1} \kappa(G_i)|V|)$, which does not increase the time complexity of a standalone ML core decomposition process without materializing the results.

2) Proof of Theorem 2:

Proof Sketch. **(Theorem 2)** Algorithm 2 is a straightforward parallelization of Algorithm 1, and its correctness is independent of the order in which the ML cores are generated. The total work remains the same as the serial version, which is, as stated in Theorem 1, $O(\prod_{i=1}^{|L|-1} \kappa(G_i)(|L| \cdot |V| + |E|))$. During the execution, a task to explore an MLC-tree node is generated only after all ML cores on the path from the root to this node have been found. Therefore, the depth of the algorithm is determined by the time when the last task is generated and the time to finish this task, which are $O(\sum_{i=1}^{|L|-1} \kappa(G_i)(|L| \cdot |V| + |E|))$ and $O(|L| \cdot |V| + |E|)$, respectively. Thus, the depth in the theorem is established, and the theorem holds. □

3) Proof of Theorem 3:

Proof. **(Theorem 3)** Let $N_0, N_1, \cdots, N_m$ be nodes on the rightmost path of N, where $N_0 = N$ and N_m is a leaf, C_i be the ML core represented by N_i, and S_i be the vertex set associated with N_i. We have $S_i = C_i - C_{i+1}$ for $0 \leq i < m$ and $S_m = C_m$. Therefore, $\bigcup_{i=0}^{m} V_i = \bigcup_{i=0}^{m-1}(C_i - C_{i+1}) \bigcup C_m = C_0$, establishing the theorem. □

4) Proof of Theorem 4:

Proof Sketch. **(Theorem 4)** The correctness of Algorithm 4 is ensured by Lemma 1 and Theorem 3. In terms of the time overhead, the total number of tree nodes visited during the execution of Procedure SEARCH and Procedure RESTORE is bounded by the height of the MLC-tree, i.e., $O(\sum_{i=1}^{|L|} \kappa(G_i))$. Considering the extra $|C|$ time to collect the target ML core C, Algorithm 4 runs in $O(\sum_{i=1}^{|L|} \kappa(G_i) + |C|)$ time. □

5) Proof of Theorem 5: We first present some foundations for the proof of Theorem 5. Let $l_c \in L$ be the layer with the maximum degeneracy, i.e., $l_c = \arg\max_{i \in L} \kappa(G_i)$, and let $k^* = \kappa(G_{l_c})$. Let $C^*_{\text{SL}} \subseteq V$ be k^*-core on layer l_c. We have that C^*_{SL} maximizes the minimum vertex degree among all possible vertex subsets on all layers. Certainly, $C^*_{\text{SL}} \in \mathcal{C}$.

Lemma 2 ([6]). $\rho(C^*) \geq \rho(C^*_{SL})$.

Next, let $\mu(S, l)$ represent the minimum vertex degree in $G_l[S]$. Define S^*_{SL} as the weighted densest subgraph among all single layers, i.e., $S^*_{\text{SL}} = \arg\max_{S \subseteq V} \max_{i \in L} \frac{w_i |E_i[S]|}{|S|}$, and l_s be the layer where S^*_{SL} exhibits that highest density.

Lemma 3 ([6]). $\mu(S^*_{SL}, l_s) \geq \frac{|E_{l_s}[S^*_{SL}]|}{|S^*_{SL}|}$.

Lemma 4. $\rho(S^*) \leq \frac{w_{l_s} |E_{l_s}[S^*_{SL}]|}{|S^*_{SL}|} |L|^{\beta}$.

We then give the proof of Theorem 5:

Proof. **(Theorem 5)** It holds that:

$$
\begin{aligned}
\rho(C^*) \geq \rho(C^*_{\text{SL}}) &\geq \max_{i \in L} \frac{w_i |E_i[C^*_{\text{SL}}]|}{|C^*_{\text{SL}}|} \cdot 1^{\beta} \geq \frac{w_{l_c} |E_{l_c}[C^*_{\text{SL}}]|}{|C^*_{\text{SL}}|} \\
&\geq \frac{1}{2} w_{l_c} \mu(C^*_{\text{SL}}, l_c) \qquad \triangleright \text{ avg. degree} > \text{min. degree} \\
&\geq \frac{1}{2} w_{l_c} \mu(S^*_{\text{SL}}, l_s) \qquad \triangleright C^*_{\text{SL}}, l_c \text{ maximize } \mu(\cdot) \\
&\geq \frac{1}{2} w_{l_c} \frac{|E_{l_s}[S^*_{\text{SL}}]|}{|S^*_{\text{SL}}|} \\
&\geq \frac{1}{2} \frac{w_{l_c}}{w_{l_s} |L|^{\beta}} \rho(S^*) \geq \frac{w^-}{2 w^+ |L|^{\beta}} \rho(S^*),
\end{aligned}
$$

which establishes Theorem 5. □

6) Proof of Theorem 6:

Proof sketch. **(Theorem 6)** Algorithm 5 returns a $\frac{w^-}{2w^+|L|^\beta}$-approximation to the optimal solution of the WDS problem. Theorem 5 establishes the correctness of Algorithm 5 and the approximation ratio. Consider the computation for a given rightmost path, the time overhead is dominated by the restoration of the per-layer edge numbers. With the basic MLC-tree, the restoration against $\mathcal{G}$ requires $O(|L| \cdot |V| + |E|)$ time, while using the edge-difference-augmented MLC-tree incurs a cost of $\kappa(G_{|L|})$ time. As the number of rightmost paths is bounded by $O(\prod_{i=1}^{|L|-1} \kappa(G_i))$, the time complexities given in the theorem hold. □

REFERENCES

[1] Boost C++ Libraries. boost/container_hash/hash.hpp - 1.77.0. Accessed on: 2023-11. [Online]. Available: https://www.boost.org/doc/libs/1_77_0/boost/container_hash/hash.hpp

[2] Y. Zhang, J. X. Yu, Y. Zhang, and L. Qin, "A fast order-based approach for core maintenance," in *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 2017, pp. 337–348.

[3] D. Liu and Z. Zou, "gcore: Exploring cross-layer cohesiveness in multi-layer graphs," *Proceedings of the VLDB Endowment*, vol. 16, no. 11, pp. 3201–3213, 2023.

[4] F. Zhang, J. Zhai, X. Shen, D. Wang, Z. Chen, O. Mutlu, W. Chen, and X. Du, "Tadoc: Text analytics directly on compression," *The VLDB Journal*, vol. 30, pp. 163–188, 2021.

[5] Z. Chen, F. Zhang, J. Guan, J. Zhai, X. Shen, H. Zhang, W. Shu, and X. Du, "Compressgraph: Efficient parallel graph analytics with rule-based compression," *Proceedings of the ACM on Management of Data*, vol. 1, no. 1, pp. 1–31, 2023.

[6] E. Galimberti, F. Bonchi, F. Gullo, and T. Lanciano, "Core decomposition in multilayer networks: Theory, algorithms, and applications," *ACM Trans. Knowl. Discov. Data*, vol. 14, no. 1, pp. 11:1–11:40, 2020.