gCore: Exploring Cross-layer Cohesiveness in Multi-layer Graphs

Dandan Liu
Harbin Institute of Technology
Harbin, Heilongjiang, China
ddliu@hit.edu.cn

Zhaonian Zou
Harbin Institute of Technology
Harbin, Heilongjiang, China
znzou@hit.edu.cn

ABSTRACT

As multi-layer graphs can give a more accurate and reliable picture of the complex relationships between entities, cohesive subgraph mining, a fundamental task in graph analysis, has been studied on multi-layer graphs in the literature. However, existing cohesive subgraph models are designated for special multi-layer graphs, such as multiplex networks and heterogeneous information networks. In this paper, we propose generalized core (gCore), a new notion of cohesive subgraph on general multi-layer graphs without any predefined constraints on inter-connections between vertices. The gCore model considers both intra-layer and cross-layer cohesiveness of vertices. Three related problems are studied in this paper including gCore search (GCS), gCore decomposition (GCD) and gCore indexing (GCI). A polynomial-time algorithm based on the peeling paradigm is proposed to solve the GCS problem. By considering the containment among gCores, a "tree of trees" data structure called KP-tree is designed for efficiently solving the GCD problem and serving as a compact storage and index of all gCores. Several advanced lossless compaction techniques including node/subtree-elimination, subtree transplant and subtree merge are proposed to help reduce the storage overhead of the KP-tree and speed up the process of solving GCD and GCI. Besides, a KP-tree-based GCS algorithm is designed, which can retrieve any gCore in linear time in the size of the gCore and the height of the KP-tree. The experiments on 10 real-world graphs verify the effectiveness of the gCore model and the efficiency of the proposed algorithms.

PVLDB Reference Format:
Dandan Liu and Zhaonian Zou. gCore: Exploring Cross-layer Cohesiveness in Multi-layer Graphs. PVLDB, 16(1): XXX-XXX, 2023.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/SSSuperDan/gCore.

1 INTRODUCTION

Motivation. Cohesive subgraph mining (CSM), as a fundamental task in graph analysis, aims at finding densely connected vertices. It has witnessed considerable applications, such as community detection/search [7], graph visualization [1], product promotion [10], and biological module discovery [41]. While CSM has been extensively studied in single-layer graphs [21], the limitations of single-layer graphs in capturing multiple types of relationships, such as diverse social relationships [9], have led to growing research interest in CSM on *multi-layer graphs* (referred to as ML-CSM) [3, 13, 19, 23, 41].

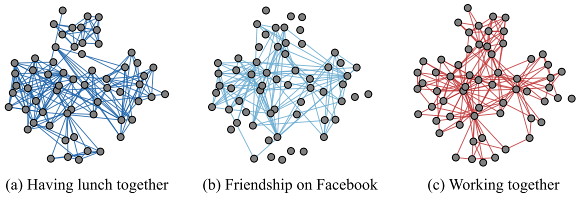


Figure 1: An example of pillar multi-layer graph extracted from the AUCS dataset [9] which consists of three layers. Each layer depicts a type of relationship between the employees in a university.

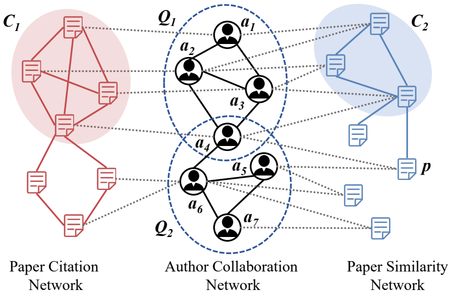


Figure 2: An example of a general multi-layer graph representing a 3-layer academic network. The layers from left to right represent the citation between papers, the collaboration between authors, and the similarity between papers. Each author is connected to his/hers published papers through cross-layer links.

A multi-layer graph is represented as a collection of interconnected layered graphs (layers for short), with each layer representing a specific type of relationship [4, 20]. There are two main kinds of multi-layer graphs. Figure 1 gives an example of the *pillar multi-layer graph* that has the same set of vertices (entities) on all layers, and every vertex in a layer has exactly one cross-layer link to its copy (mirror) on every other layer. In contrast, a *general multi-layer graph*, as depicted in Figure 2, allows different sets of vertices in each layer, and any vertex in a layer has zero to many cross-layer links to the vertices on every other layer.

Based on the multi-faceted relationships captured by the multi-layer graphs, ML-CSM enables the discovery of more reliable cohesive subgraphs [29, 41]. For instance, we are interested in cohesive author groups in the general multi-layer graph depicted in Figure 2. Obviously, vertices in Q_1 form a cohesive group in the author layer. Moreover, their cross-layer neighbors, representing the papers they have published, also exhibit dense connections in terms of both similarity and citation relationships. This information complements the existing author groups and enhances their cohesiveness.

Proceedings of the VLDB Endowment, Vol. 16, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

Existing work investigates ML-CSM on a special *pillar* multi-layer graph model named **m**ulti**p**lex **n**etworks (MPN) [12, 13, 41] and a typical *general* multi-layer graph model termed **h**eterogeneous **i**nformation **n**etworks (HIN) [11, 18, 38]. MPNs are commonly employed when diverse relationships among entities of the same type are considered, while HINs are used to study ML-CSM in the context of complex relationships between heterogeneous entities.

Unfortunately, there is currently a lack of cohesive subgraph models and algorithms that can effectively handle a generic scenario involving relationships between both homogeneous entities and heterogeneous entities, i.e., on a **g**eneral **m**ulti-layer **g**raph (GMG), as illustrated in Figure 2.

Prior Work. In the literature, two major types of cohesive subgraph models have been proposed for solving ML-CSM on MPNs, namely *cross-layer quasi-clique* [5, 29, 39] and *multi-layer core* [12, 23, 41]. They can be seen as simple extensions from the classical γ-quasi-clique [26] model and the k-core [31] model in single-layer graphs. Specifically, a cross-layer quasi-clique (resp. multi-layer core) is defined as a subset of vertices that forms a γ-quasi-clique (resp. k-core) in every layer. Certainly, such concepts are not suitable for GMGs where different layers may have different vertex sets.

A well-known branch of ML-CSM models proposed for HINs, including the $(k, \mathcal{P})$-core [11] and the (k, Ψ)-NMC [18], is based on *meta-paths*, which are sequences of vertex types and edge types between two given vertex types. Meta-paths define new adjacency relationships between vertices. For example, in the GMG shown in Figure 2, authors a_4 and a_5 are considered adjacent under a commonly used meta-path $\mathcal{P}^*$ = author-paper-author as they both connect to the paper p. The $(k, \mathcal{P})$-core [11] and the (k, Ψ)-NMC [18] find the k-core and the multi-layer core, respectively, based on the adjacency relationships defined by $\mathcal{P}$ and the set of meta-paths Ψ. However, the following two limitations of these models hinder their applicability in solving ML-CSM in GMGs: (1) Both the two models overlook the most direct and informative relationships between homogeneous entities, such as the co-authoring among authors and the similarity between papers, as shown in Figure 2. (2) Meta-path instances only reflect the connections between entities under a specific pattern, lacking the ability to capture the cohesiveness among intermediate entities. When applying $\mathcal{P}^*$ to the author collaboration and paper similarity layers in Figure 2, it yields the same adjacency pattern for vertices in Q_1 and Q_2. The information on the cohesiveness of their connected papers, which is shown can enhance the cohesiveness among authors, is completely hidden from the higher-level models and algorithms.

Another type of ML-CSM model designed for HINs, namely *relational community* [16], finds a vertex set that satisfies a collection of user-specified constraints describing "each vertex of type A has $\geq k$ neighbors of type B". However, it's challenging for users unfamiliar with the HIN schema to provide meaningful relational constraints [18]. Additionally, as the constraint is imposed on a pair of vertex types, it fails to distinguish multiple relationships between homogeneous types of entities. For example, the constraint cannot express the requirement that each paper has $\geq k_1$ neighbors in the similarity layer and meanwhile $\geq k_2$ neighbors in the citation layer.

Solution. In this paper, we propose a new cohesive subgraph model called *generalized core (gCore)* to solve the ML-CSM problem on general multi-layer graphs, which overcomes the limitations of existing models discussed before. To obtain reliable and robust cohesive subgraphs, we expect the vertices to show cohesiveness in each layer [17, 41]. A thorny problem is to define the cohesiveness of vertices on different layers based on the many-to-many cross-layer mappings between vertices. Our idea is to use a *fraction* [40] of cross-layer neighbors of a vertex to represent its engagement in different layers. Combined with the extension scheme used in cross-layer quasi-cliques [17, 29] and multi-layer cores [12], we have the gCore model described as below. Suppose there are l layers in a general multi-layer graph and the l-th layer is of users' interest. Let $k_1, k_2, \ldots, k_l \in \mathbb{N}$ and $p_1, p_2, \ldots, p_{l-1} \in [0, 1]$. A subset Q of vertices on the l-th layer forms a gCore if 1) Q is a k_l-core on the l-th layer; 2) in each other i-th layer, there exists a k_i-core $Q' \subseteq N(Q)$ such that every vertex in Q has at least a fraction p_i of neighbors on the i-th layer participating Q', where $N(Q)$ is the set of all vertices in the i-th layer that link to vertices in Q; and 3) Q is maximal.

Let us see an example. In the GMG shown in Figure 2, we number the layers from left to right as 1, 3, and 2. Given $(k_1, k_2, k_3) = (3, 2, 2)$ and $(p_1, p_2) = (1, 0.5)$, Q_1 is a gCore. Specifically, Q_1 itself forms a 2-core in the author layer. C_1 and C_2 are a 3-core and 2-core on the paper citation and similarity layers, respectively, covering all and at least a half of cross-layer neighbors of each vertex in Q_1. Obviously, the gCore ensures the cohesiveness on both the direct interactions between authors and each possible relationship between related papers. It is worth noticing that Q_2 can be well excluded by gCore under the above parameter settings.

Application. Here are typical applications of the gCore model: (1) *Collaboration analysis.* On an academic multi-layer network like Figure 2, gCore facilitates the identification of cohesive groups of authors who have collaborated on a series of similar papers, possibly on the same topic, and have often cited each other. These groups are well-suited to be invited to present tutorials on the topic and share their related papers. (2) *Recommendation/Product promotion.* E-commerce platforms often maintain HINs of products and authors [11], while social media platforms like Last.fm (https://www.last.fm/) have bipartite data describing the 'like' relationships between users and shared items. By enriching the existing HINs with additional homogeneous relationships like friendships or similarities between users, as well as similarities between products or items, gCore can be utilized to discover groups of users who are not only friends but also share similar preferences in products or items. This information can be leveraged to boost sales or user engagement by recommending products purchased or items liked by users in the group to other users within the same group. (3) *Biological analysis.* Diseases can exhibit similarities in various ways, such as sharing a significant number of therapies [8] or having commonly associated pathways [22]. Besides, communities of genes can also unveil similarities between diseases [34]. By representing these relationships in a GMG, gCore can be employed to identify meaningful disease clusters, providing new insights into disease etiology, classification, and shared biological mechanisms [22].

Technical Contributions. To address the gap in solving the ML-CSM problem on GMGs, we propose the gCore model and fully investigate its properties. One notable property is that given vectors $\mathbf{k} = (k_1, k_2, \ldots, k_l)$ and $\mathbf{p} = (p_1, p_2, \ldots, p_{l-1})$, there exists a unique gCore, which we therefore refer to as the $(\mathbf{k}, \mathbf{p})$-core.

Based on the gCore model, we study three related problems in this paper, namely *gCore search (GCS)*, *gCore decomposition (GCD)* and *gCore indexing (GCI)*. GCS finds the $(\mathbf{k}, \mathbf{p})$-core in a l-layer GMG for given vectors $\mathbf{k}$ and $\mathbf{p}$. This is crucial for retrieving cohesive subgraphs with particular features. GCD finds all distinct nonempty gCores in a GMG, which is essential for exploring the structure of a GMG. GCI constructs an index to speed up GCS.

To solve the GCS problem, we propose a polynomial-time algorithm based on the *vertex peeling* paradigm.

To address the GCD problem, we propose a "tree of trees" data structure called *KP-tree* to organize all gCores in a systematic way by leveraging their containment relationships. The KP-tree determines a DFS-order generation of all gCores which reduces redundant computations and enables fast identification of empty gCores.

Moreover, the KP-tree derives an index structure to support fast gCore retrieval. Based on this index, a more efficient algorithm is proposed to solve the GCS problem, which runs in linear time in the size of the queried gCore and the height of the KP-tree. Considering the redundancy of information in the KP-tree index, we design a series of lossless compaction schemes, including node/subtree elimination, subtree transplant, and subtree merge, to further save storage and improve index construction efficiency.

Extensive experiments have been conducted to evaluate the gCore model and the proposed algorithms. The results show that: (1) Vertices in the gCore attain more closeness with each other compared with those in the k-core and other variations of the existing ML-CSM models adapted to suit GMGs. (2) The index helps improve the efficiency of gCore search by up to 1–4 orders of magnitude; (3) The index compaction techniques significantly reduce both the index construction time, which includes the time to solve the GCD problem, and the storage overhead of the index.

2 RELATED WORK

In this section, we provide a brief overview of existing work on the ML-CSM problem.

CSM on MPNs. To characterize cohesive subgraphs on an MPN, existing work always uses a unified classic cohesive subgraph model to restrict the cohesiveness of vertices in some/all layers. Wang et al. [36] and Zeng et al. [39] studied to mine maximal frequent cliques and γ-quasi-cliques in a series of graphs. Pei et al. [29] and Jiang et al. [17] extended the notion of quasi-clique and introduced the cross-layer quasi-clique model and the frequent cross-layer quasi-clique model. Due to the #P-complexity of enumerating all (frequent) cross-layer quasi-cliques, exact branch and bound approaches with several pruning methods are proposed. Boden et al. [5] introduced an MLCS cluster model for MPNs with edge labels, which is a variant of cross-layer quasi-clique with consideration of similarities between edge labels. A best-first search approach is presented in [5] to find qualified MLCS clusters with low redundancy.

Due to the high computational cost and low flexibility in characterizing large cohesive subgraphs of the cross-layer quasi-cliques, Zhu et al. [41] proposed a notion of d-coherent core (d-CC), which requires a vertex subset to form a d-core in a given subset of layers. Three approximation algorithms with provable guarantees are proposed in [41] to extract a fixed number of d-CCs that attain the largest diversity. Liu et al. [23] studied the d-CC decomposition problem. Galimberti et al. [12] studied the multi-layer core model, another extension of k-core. It allows using varying k in different layers.Three decomposition algorithms relying on different search orders are given in [12]. Hashemi et al. [13] introduced a relaxed version of the d-CC model called (k, λ)-FirmCore, and studied the FirmCore decomposition problem. Huang et al. [15] and Behrouz et al. [3] further combined the extension scheme of the d-coherent core and the (k, λ)-FirmCore with the k-truss model, respectively. All the above models rely on the one-to-one cross-layer mappings between vertices, which are hence inapplicable to GMGs.

CSM on HINs. Fang et al. [10] surveyed the existing CSM models and algorithms designed for HINs. Here, we review several representative works. Fang et al. [11] defined a $(k, \mathcal{P})$-core model based on a given symmetric meta-path $\mathcal{P}$, which requires each vertex in a $(k, \mathcal{P})$-core is connected to at least k vertices through instances of $\mathcal{P}$. Two variants of the $(k, \mathcal{P})$-core are also studied in [11] that require the meta-path instances contributing to k are vertex-disjoint and edge-disjoint. Yang et al. [38] combined the extension scheme of the $(k, \mathcal{P})$-core and the vertex-disjoint $(k, \mathcal{P})$-core with the k-truss model. Jian et al. [16] proposed another extension of k-core called relational community. It allows using a series of meta-paths $\mathcal{P}$ of fixed length 2 and integers k specific to each $\mathcal{P}$. A relational community may contain heterogeneous vertices. Hu et al. [14] designed an extension of the clique model called m-Clique based on a given meta-subgraph. Due to the challenge of selecting meaningful meta-paths or relational constraints, Jiang et al. [18] introduced a (k, Ψ)-NMC model, where Ψ is a set of non-nested meta-paths. A set of vertices is a (k, Ψ)-NMC if it forms a $(k, \mathcal{P})$-core for each meta-path $\mathcal{P} \in \Psi$. Algorithms to search the (k, Ψ)-NMC containing a given set of query vertices while maximizing the size of Ψ are proposed in [18]. However, as discussed in Section 1, the above models cannot be directly applicable to GMGs that involve multiple types of relationships between homogeneous entities.

3 PRELIMINARIES

In this section, we introduce the basics and concepts of the GMG model, and then present our generalized core model along with its properties. Finally, we outline the problems addressed in this paper.

3.1 The General Multi-layer Graph Model

A simple graph $G = (V, E)$ is a pair where V is the set of vertices and E is the set of edges. By joining multiple simple graphs via the connections among them, we have a general multi-layer graph:

Definition 3.1 ([25]). A *general multi-layer graph* (GMG) is a pair $\mathcal{M} = (\mathcal{G}, C)$, where $\mathcal{G} = \{G_1, G_2, \ldots, G_l\}$ is a set of simple graphs (also called *layers*) $G_i = (V_i, E_i)$, where $i \in \{1, 2, \cdots, l\}$, and $C = \{E_{i,j} | 1 \leq i < j \leq l\}$ is the collection of sets of edges $E_{i,j} \subseteq V_i \times V_j$ linking vertices of G_i to vertices of G_j.

We denote the edges in $E_1, E_2, \ldots, E_l$ as "*intra-layer edges*" and the edges in $E_{i,j}$ as "*cross-layer edges*". We use $E(\mathcal{G})$ and $E(C)$ to denote the set of all intra-layer edges and all cross-layer edges, respectively, i.e., $E(\mathcal{G}) = \bigcup_{i=1}^{l} E_i$ and $E(C) = \bigcup_{E_{i,j} \in C} E_{i,j}$.

Obviously, pillar multi-layer graphs are special cases of GMGs where all layers have identical vertex sets and a cross-layer edge only exists between two copies of a vertex lying in different layers.

In GMGs, a vertex $v \in V_i$ can have both intra-layer and cross-layer neighbors. The set of neighbors of v in G_j is denoted by $N_j(v)$. If $j = i$, the vertices in $N_j(v)$ are called *intra-layer neighbors*; otherwise, they are called *cross-layer neighbors*. The degree of v in G_j is denoted as $deg_j(v)$, where $deg_j(v) = |N_j(v)|$ is the *intra-layer degree* of v if $j = i$, and the *cross-layer degree* otherwise.

A GMG $\mathcal{M}' = (\mathcal{G}', \mathcal{C}')$ is a subgraph of $\mathcal{M}$ if $|\mathcal{G}| = |\mathcal{G}'|$, G'_i is a subgraph of G_i for $G'_i \in \mathcal{G}'$, and $E'_{i,j} \subseteq E_{i,j}$ for $E'_{i,j} \in \mathcal{C}'$. Given a simple graph $G = (V, E)$, the subgraph of G *induced* by $Q \subseteq V$ is $G[Q] = (Q, E[Q])$, where $E[Q] \subseteq E$ is the set of edges with both endpoints in Q. Similarly, let $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_l\}$, where $Q_i \subseteq V_i$ for $1 \leq i \leq l$, the subgraph of $\mathcal{M}$ induced by $\mathcal{Q}$ is $\mathcal{M}[\mathcal{Q}] = (\mathcal{G}[\mathcal{Q}], \mathcal{C}[\mathcal{Q}])$, where $\mathcal{G}[\mathcal{Q}] = \{G_1[Q_1], G_2[Q_2], \ldots, G_l[Q_l]\}$ and $\mathcal{C}[\mathcal{Q}] = \{E_{i,j}[Q_i, Q_j] | 1 \leq i < j \leq l\}$ with $E_{i,j}[Q_i, Q_j] \subseteq E_{i,j}$ being the set of edges having one endpoint in Q_i and the other in Q_j.

Besides, the presence of cross-layer edges in a GMG enable us to define cross-layer induced subgraphs. For a vertex subset $Q_i \subseteq V_i$ and $j \neq i$, let $Q_j = \bigcup_{v \in Q_i} N_j(v)$ be Q_i's cross-layer neighbors in G_j. The induced subgraph $G_j[Q_j]$ is called the *cross-layer subgraph* of G_j induced by Q_i, denoted by $G_j[Q_i]$.

3.2 gCores in General Multi-layer Graphs

As stated in Section 1, a generalized core (gCore) Q on the layer of users' interest, say G_i, is expected to show cohesiveness under various relationships, including the direct one captured by G_i and the indirect ones represented by $E_{i \to j}$ and G_j for $j = 1, 2, \cdots, l$ and $j \neq i$. The former can be characterized by the elegant k-core model [31], which requires each vertex $v \in Q$ to be adjacent to at least k vertices in $G_i[Q]$, i.e., the degree of v in $G_i[Q]$ is at least k. To characterize the latter, we extend the concept of *fraction* proposed in [40] and rephrase it as *neighbor coverage fraction* in Definition 3.2.

Definition 3.2. For $v \in V_i$ and $Q_j \subseteq V_j$, the fraction of v's neighbors in G_j falling in Q_j, i.e., $\phi(v, Q_j) = \frac{|N_j(v) \cap Q_j|}{|N_j(v)|}$, is called the *neighbor coverage fraction* of v within Q_j.

A higher value of $\phi(v, Q_j)$ indicates that Q_j covers more neighbors of v in G_j, thereby capturing more information about v's cross-layer neighborhood. Based on Definition 3.2, we can define the concept of generalized core (gCore) on GMGs.

Definition 3.3. Given an l-layer graph $\mathcal{M}$, a specific layer G_i, l thresholds $k_1, k_2, \ldots, k_l \in \mathbb{N}$ and $p_{i,j} \in [0, 1]$ for $j = 1, 2, \ldots, l$ and $j \neq i$, a vertex subset $Q_i \subseteq V_i$ is a *generalized core (gCore)* in $\mathcal{M}$ if

(1) Q_i is a k_i-core in G_i.
(2) For $j = 1, 2, \ldots, l$ and $j \neq i$, there exists a nonempty k_j-core Q_j in $G_j[Q_i]$ such that $\phi(v, Q_j) \geq p_{i,j}$ for all $v \in Q_i$.
(3) None of the proper supersets of Q_i satisfies (1) and (2).

Without loss of generality, we suppose the selected layer in Definition 3.3 is G_l in the rest of the paper. For simplicity, let $\mathbf{k} = (k_1, k_2, \ldots, k_l)$ and $\mathbf{p} = (p_{l,1}, p_{l,2}, \ldots, p_{l,l-1})$. A gCore defined with respect to vectors $\mathbf{k}$ and $\mathbf{p}$ is referred to as a $(\mathbf{k}, \mathbf{p})$-core.

Example. Figure 3 shows a 3-layer GMG, where G_2 is of users' interest. Let $\mathbf{k} = (3, 3, 3)$, $\mathbf{p}_1 = (0, 0)$, $\mathbf{p}_2 = (1/2, 0)$, $\mathbf{p}_3 = (1/2, 2/3)$. The entire vertex set of G_2, V_2, forms the $(\mathbf{k}, \mathbf{p}_1)$-core as $\mathbf{p}_1$ does not impose any constraints on the cohesiveness of the vertices on G_0 and G_1, and V_2 itself is a 3-core. The $(\mathbf{k}, \mathbf{p}_2)$-core is $\{1, 2, 3, 4, 5, 6, 7, 8\}$. Vertices $\{9, 10\}$ are excluded from the $(\mathbf{k}, \mathbf{p}_2)$-core because they have no cross-layer neighbors in G_0. $Q = \{1, 2, 3, 4\}$ forms the $(\mathbf{k}, \mathbf{p}_3)$-core. Vertices $\{22, 23, 24, 25, 26\}$ on G_0 and vertices $\{13, 14, 15, 16, 17\}$ on G_1 are both 3-cores and cover at least $1/2$ and $2/3$ neighbors of each vertex in Q, respectively.

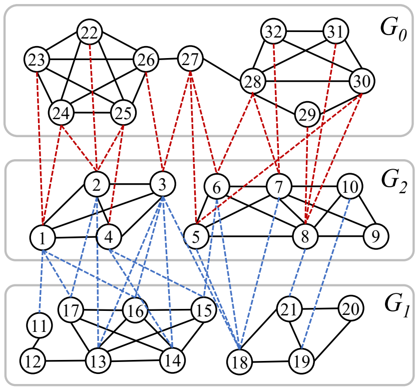


Figure 3: An example of a general multi-layer graph (GMG).

Generalization. The gCore model is a generalization of the multi-layer $\mathbf{k}$-core model [12] and the d-CC model [41] proposed for MPNs. Given an l-layer multi-layer graph $\mathcal{M}$, if $\mathcal{M}$ is an MPN, the $(\mathbf{k}, \mathbf{p})$-core of $\mathcal{M}$, where $\mathbf{p} = \mathbf{1}^{l-1}$, is the multi-layer $\mathbf{k}$-core on $\mathcal{M}$. Moreover, if we consider a subset L of layers, the gCore w.r.t. $\mathbf{p} = \mathbf{1}^{l-1}$ and $\mathbf{k} = [k_i]_{1 \leq i \leq l}$, where $k_i = d$ if $G_i \in L$ and $k_i = 0$ otherwise, corresponds to the d-CC on L.

Properties. The concept of gCore has several elegant properties.

Property 1. *Given* $\mathbf{k}$ *and* $\mathbf{p}$*, the* $(\mathbf{k}, \mathbf{p})$*-core is unique.*

As a notation, let $\mathbf{x}$ and $\mathbf{y}$ be two vectors of the same dimension, we say $\mathbf{x} \leq \mathbf{y}$ if $\mathbf{x}[i] \leq \mathbf{y}[i]$ for every dimension i, and we say $\mathbf{x} < \mathbf{y}$ if $\mathbf{x} \leq \mathbf{y}$ and $\mathbf{x} \neq \mathbf{y}$.

Property 2. *Given* $\mathbf{k}_1$*,* $\mathbf{k}_2$ *and* $\mathbf{p}$*, if* $\mathbf{k}_1 \leq \mathbf{k}_2$*, the* $(\mathbf{k}_2, \mathbf{p})$*-core is a subset of the* $(\mathbf{k}_1, \mathbf{p})$*-core.*

Property 3. *Given* $\mathbf{k}$*,* $\mathbf{p}_1$ *and* $\mathbf{p}_2$*, if* $\mathbf{p}_1 \leq \mathbf{p}_2$*, the* $(\mathbf{k}, \mathbf{p}_2)$*-core is a subset of the* $(\mathbf{k}, \mathbf{p}_1)$*-core.*

Due to limited space, we leave the proofs of all the properties, lemmas, and theorems in Appendix M of the full paper [24].

3.3 Problem Formulation

This paper studies the following three problems related to the proposed gCore model on general multi-layer graphs:

(1) **gCore Search (GCS).** Given an l-layer graph $\mathcal{M}$, $\mathbf{k} \in \mathbb{N}^l$ and $\mathbf{p} \in [0, 1]^{l-1}$, find the $(\mathbf{k}, \mathbf{p})$-core in $\mathcal{M}$.

(2) **gCore Decomposition (GCD).** Given an l-layer graph $\mathcal{M}$, enumerate all nonempty and distinct gCores in $\mathcal{M}$.

(3) **gCore Indexing (GCI).** Given an l-layer graph $\mathcal{M}$, store all nonempty gCores in $\mathcal{M}$ in a compact data structure to support fast retrieval of the $(\mathbf{k}, \mathbf{p})$-core for given $\mathbf{k}$ and $\mathbf{p}$.

4 GCORE SEARCH (GCS)

Given an l-layer graph $\mathcal{M}$, $\mathbf{k} \in \mathbb{N}^l$ and $\mathbf{p} \in [0, 1]^{l-1}$, we propose a polynomial-time algorithm called GCS to find the $(\mathbf{k}, \mathbf{p})$-core in $\mathcal{M}$. The algorithm follows the *vertex peeling paradigm* that has

Algorithm 1 GCS (gCore Search)

```
Input: An $l$-layer graph $\mathcal{M}$, $\mathbf{k} \in \mathbb{N}^l$ and $\mathbf{p} \in [0,1]^{l-1}$
Output: The $(\mathbf{k},\mathbf{p})$-core in $\mathcal{M}$
 1: $Q_l \leftarrow V_l$                                  ▷ $V_l$ is the vertex set of $G_l$
 2: repeat
 3:     $Q_{before} \leftarrow Q_l$
 4:     $Q_l \leftarrow \text{peel}(G_l[Q_l], \mathbf{k}[l])$
 5:     for $i \leftarrow 1, 2, \ldots, l-1$ do
 6:         $Q_i \leftarrow \text{peel}(G_i[Q_l], \mathbf{k}[i])$
 7:         for $v \in Q_l$ do
 8:             if $\phi(v, Q_i) < \mathbf{p}[i]$ then
 9:                 $Q_l \leftarrow Q_l - \{v\}$
10: until $Q_l = Q_{before}$
11: return $Q_l$
```

been successfully used by the existing core search/decomposition algorithms [2], that is, it iteratively removes from G_l the vertices that violate Constraint (1) or (2) specified in Definition 3.3.

Algorithm 1 presents the pseudocode of GCS. We use Q_l to keep the vertices remaining in G_l during vertex peeling. Initially, Q_l is set to the vertex set V_l of G_l in line 1. The **repeat** loop in lines 2–10 performs the peeling process on Q_l as follows. In line 3, a copy of Q_l is saved in Q_{before} to support checking the variation of Q_l in line 10. In line 4, we call the peel function to iteratively remove vertices with degrees less than $\mathbf{k}[l]$ from $G_l[Q_l]$ (i.e., violating Constraint (1) in Definition 3.3). The output of peel is the $\mathbf{k}[l]$-core in $G_l[Q_l]$, and is updated to Q_l. Then, for each other layer G_i, where $1 \le i < l$, we compute the cross-layer subgraph of G_i induced by Q_l, i.e., $G_i[Q_l]$, and call peel to obtain the $\mathbf{k}[i]$-core in $G_i[Q_l]$. The result is updated to Q_i. In lines 7–9, we check the neighbor coverage fraction $\phi(v, Q_i)$ for each vertex $v \in Q_l$. If $\phi(v, Q_i) < \mathbf{p}[i]$, v violates Constraint (2) in Definition 3.3, so we remove v from Q_l (line 9). If Q_l is changed in this iteration, i.e., $Q_l \neq Q_{before}$, the **repeat** loop continues to peel more vertices; Otherwise, no further vertices can be peeled from Q_l, the algorithm returns Q_l and terminates.

Theorem 4.1. *The output of Algorithm 1 is the* $(\mathbf{k},\mathbf{p})$*-core in* $\mathcal{M}$.

We implement Algorithm 1 based on a set of well-designed arrays (see Appendix B of the full paper [24]), leading to a time complexity of $O(|\mathcal{M}|+l|V_l|)$ and a space complexity of $O(l \cdot V_{max})$, where $|\mathcal{M}| = \sum_{i=1}^{l} |V_i| + |E(\mathcal{G})| + |E(C)|$ and $V_{max} = \max_{1 \le i \le l} |V_i|$. Complexity analysis for all algorithms is given in Appendix N of [24].

5 GCORE DECOMPOSITION (GCD)

In this section, we propose algorithms for the gCore decomposition (GCD) problem. As with the multi-layer cores [12], the number of distinct gCores can be exponential in the number of layers, and not all of them are nested into one another. Hence, we cannot expect to apply the vertex peeling paradigm used in core decomposition on simple graphs to find all gCores.

To handle this problem, we first present a naïve solution in Section 5.1. By exploiting the containment relationships among gCores as described in Properties 2 and 3, a tree-based structure called *KP-tree* is introduced to model the search space of the GCD problem in Section 5.2. The KP-tree structure determines a good order for fast computing all gCores.

5.1 Naïve GCD Algorithm

A naïve solution to the GCD problem consists of two phases. In phase 1, we enumerate all $(\mathbf{k},\mathbf{p})$ pairs such that the $(\mathbf{k},\mathbf{p})$-core is nonempty. In phase 2, we use the GCS algorithm (Algorithm 1) to compute the $(\mathbf{k},\mathbf{p})$-core for each $(\mathbf{k},\mathbf{p})$ pair enumerated in phase 1. Obviously, phase 1 is the key to the algorithm design.

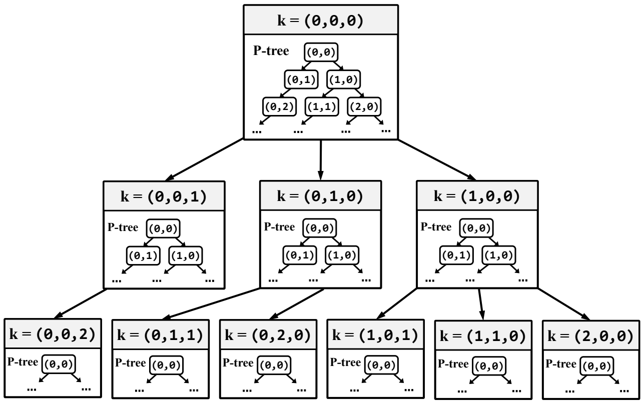


Figure 4: The structure of KP-tree of a 3-layer graph.

All elements of $\mathbf{k}$ are integers. For $i = 1, 2, \ldots, l$, we have $0 \le \mathbf{k}[i] \le \kappa(G_i)$, where $\kappa(G_i)$ is the degeneracy of G_i, the largest integer k such that the k-core of G_i is nonempty. If $\mathbf{k}[i] > \kappa(G_i)$ for some $i \in \{1, 2, \ldots, l\}$, the $\mathbf{k}[i]$-core of G_i is certainly empty, thereby violating Constraint (1) or (2) specified in Definition 3.3. Therefore, we only need to consider possible values for $\mathbf{k}$ ranging from $(0, 0, \ldots, 0)$ to $(\kappa(G_1), \kappa(G_2), \ldots, \kappa(G_l))$.

Every element of $\mathbf{p}$ is a real number in $[0, 1]$. It is definitely infeasible to enumerate all values of $\mathbf{p}$. However, we found that it is sufficient to choose $\mathbf{p}[i]$ from the finite set F_i defined as below.

Lemma 5.1. *For any* $(\mathbf{k},\mathbf{p})$*-core* Q *in the multi-layer graph* $\mathcal{M}$*, there exists a vector* $\hat{\mathbf{p}}$ *with* $\hat{\mathbf{p}}[i] \in F_i$ *for* $i = 1, 2, \cdots, l-1$ *such that the* $(\mathbf{k},\hat{\mathbf{p}})$*-core in* $\mathcal{M}$ *is identical to* Q*, where*

$$F_i = \left\{ \frac{j}{deg_i(v)} \middle| v \in V_l, j = 0, 1, \ldots, deg_i(v) \right\}. \tag{1}$$

After computing each F_i against Equation (1), we can then generate all possible values for $\mathbf{p}$.

The correctness of the two-phase naïve solution is obvious. For the sake of the limited space, the pseudocode is given in Appendix C of the full paper [24]. The time complexity of the naïve approach is $O(\prod_{i=1}^{l} \kappa(G_i) \cdot \prod_{i=1}^{l-1} |F_i| \cdot (|\mathcal{M}| + l|V_l|) + \sum_{i=1}^{l-1} d_i^2 \log d_i)$, and the space complexity is $O(l \cdot (\prod_{i=1}^{l} \kappa(G_i) + \prod_{i=1}^{l-1} |F_i| + V_{max}))$, where d_i is the maximum cross-layer degree of vertices in V_l on layer G_i.

The naïve approach has two main disadvantages. First, it computes the $(\mathbf{k},\mathbf{p})$-core for all enumerated $(\mathbf{k},\mathbf{p})$ pairs, although many of them are empty. Second, each $(\mathbf{k},\mathbf{p})$-core is independently computed by Algorithm 1 from scratch. A large number of repeated computations are carried out for different $(\mathbf{k},\mathbf{p})$ pairs.

5.2 GCD Algorithm based on KP-Trees

To overcome the disadvantages of the naïve approach, we design a "tree of trees" data structure called *KP-tree* to represent the search space of the GCD problem. We will show later that by traversing a KP-tree, it is possible to fast identify empty gCores and reduce repeated computations when computing different gCores.

5.2.1 **KP-Trees.** As illustrated in Figure 4, the overall structure of a KP-tree is a tree. Each node in the KP-tree is associated with an l-dimensional vector $\mathbf{k} \in \mathbb{N}^l$. We call the node associated with a vector $\mathbf{k}$ the $\mathbf{k}$-*node*. The root of the KP-tree is the $\mathbf{0}^l$-node. In the KP-tree, the $\mathbf{k}$-node is the parent of the $\mathbf{k}'$-node if and only if $\mathbf{k}'$ is an immediate suffix successor of $\mathbf{k}$ as defined below.

Definition 5.2. Let $\mathbf{x}$ and $\mathbf{y}$ be two integer vectors of the same dimension. $\mathbf{y}$ is an *immediate suffix successor* of $\mathbf{x}$ if

(1) $\mathbf{x}$ and $\mathbf{y}$ are different in exactly one element, say $\mathbf{x}[i]$ and $\mathbf{y}[i]$;
(2) $\mathbf{x}[i] + 1 = \mathbf{y}[i]$;
(3) $\mathbf{x}[j] = \mathbf{y}[j] = 0$ for all $j > i$.

Obviously, if $\mathbf{y}$ is an immediate suffix successor of $\mathbf{x}$, we have $\mathbf{x} < \mathbf{y}$. Let $end0(\mathbf{k})$ be the number of consecutive zeros at the end of a vector $\mathbf{k}$. It's easy to see that the $\mathbf{k}$-node in the KP-tree has $end0(\mathbf{k}) + 1$ children if $\mathbf{k} \neq \mathbf{0}^l$ and l children if $\mathbf{k} = \mathbf{0}^l$.

Inside every node of the KP-tree is nested another tree called *P-tree*. Each node in the *P*-tree is associated with an $(l-1)$-dimensional real vector $\mathbf{p} \in [0, 1]^{l-1}$, and the node is called the $\mathbf{p}$-node. In fact, as the i-th element of $\mathbf{p}$ is chosen from the set F_i of fractional numbers formulated in Equation (1), we actually store in each $\mathbf{p}[i]$ the index of the corresponding fractional value in F_i after sorting it in increasing order. It helps to define the P-tree and save storage. Specifically, in a P-tree, the root is the $\mathbf{0}^{l-1}$-node, the $\mathbf{p}$-node is the parent of the $\mathbf{p}'$-node if and only if $\mathbf{p}'$ is an immediate suffix successor of $\mathbf{p}$. Thus, the $\mathbf{p}$-node has $end0(\mathbf{p}) + 1$ children if $\mathbf{p} \neq \mathbf{0}^{l-1}$ and $l - 1$ children if $\mathbf{p} = \mathbf{0}^{l-1}$.

The following lemma ensures that the KP-tree is a systematic organization of gCores: a $\mathbf{p}$-node nested in a $\mathbf{k}$-node represents a distinct $(\mathbf{k}, \mathbf{p})$ pair that uniquely corresponds to the $(\mathbf{k}, \mathbf{p})$-core.

Lemma 5.3. *For each possible value of* $\mathbf{k}$*, there is exactly one* $\mathbf{k}$*-node in the KP-tree. And in every* $\mathbf{k}$*-node, for each possible value of* $\mathbf{p}$*, there is exactly one* $\mathbf{p}$*-node in the P-tree nested in the* $\mathbf{k}$*-node.*

5.2.2 KP-tree-based GCD Algorithm.

It's easy to see that both KP-tree and each P-tree inside the KP-tree maintain an order '<' between vectors associated with a pair of father and child nodes. The following lemma helps leverage these orders to overcome the disadvantages of the naïve method.

Lemma 5.4. *Given a multi-layer graph* $\mathcal{M}$*, for any pair of the* $(\mathbf{k}, \mathbf{p})$*-core* Q *and the* $(\mathbf{k}', \mathbf{p}')$*-core* Q' *in* $\mathcal{M}$*,* $\mathcal{M}_{\mathbf{k}'}[Q']$ *is a subgraph of* $\mathcal{M}_{\mathbf{k}}[Q]$ *if* $\mathbf{k} \leq \mathbf{k}'$ *and* $\mathbf{p} \leq \mathbf{p}'$*, where* $\mathcal{M}_{\mathbf{k}}[Q]$ *is the subgraph of* $\mathcal{M}$ *induced by* $Q = \{Q_1, Q_2, \cdots, Q_l\}$*,* Q_i *is the* $\mathbf{k}[i]$*-core of* $G_i[Q]$ *for* $1 \leq i < l$*, and* $Q_l = Q$*.*

The lemma has two implications: (I1) If $Q = \emptyset$, we also have $Q' = \emptyset$, so it is unnecessary to compute Q' by the GCS algorithm (Algorithm 1); and (I2) otherwise, Q' can be computed on the subgraph $\mathcal{M}_{\mathbf{k}}[Q]$ instead of on the entire $\mathcal{M}$. Implication I1 helps to fast identify empty gCores, and Implication I2 helps to reduce repeated computations when computing different gCores.

The two implications and the orders between vectors associated with father and child nodes in the KP-tree and P-trees pave the way to a depth-first-search (DFS) order generation of all gCores, which computes gCores during the DFS on the KP-tree and the DFS on the P-tree nested in each visited $\mathbf{k}$-node (Algorithm 2). Notice that the function GCS used in line 5 of Algorithm 2 needs to be simply adapted to return Q_l as well as $Q_1, Q_2, \ldots, Q_{l-1}$, where each Q_i for $1 \leq i < l$ is obtained in line 6 of Algorithm 1. The pseudocode is clear, and we leave the detailed description and related discussions in Appendix D of the full paper [24].

Algorithm 2 GCD+ (KP-tree-based gCore Decomposition)

```
Input: An $l$-layer graph $\mathcal{M}$
Output: The collection $R$ of all nonempty gCores
 1: $R \leftarrow \emptyset$
 2: KPTreeDFS($\mathcal{M}, \mathbf{0}^l, R$)
 3: return $R$
 4: procedure PTreeDFS($\mathcal{M}, \mathbf{k}, \mathbf{p}, R$)
 5:     $\{Q_1, Q_2, \ldots, Q_l\} \leftarrow$ GCS($\mathcal{M}, \mathbf{k}$, ToFrac($\mathbf{p}$))
 6:     if $Q_l \neq \emptyset$ then
 7:         $R \leftarrow R \cup \{(Q_l, \mathbf{k}, \mathbf{p})\}$
 8:         for $i \leftarrow l-1, l-2, \ldots, l - end0(\mathbf{p}) - 1$ do
 9:             if $i > 0$ and $\mathbf{p}[i] < |F_i|$ then
10:                 $\mathbf{p}' \leftarrow \mathbf{p}$
11:                 $\mathbf{p}'[i] \leftarrow \mathbf{p}'[i] + 1$
12:                 PTreeDFS($\mathcal{M}[\{Q_1, Q_2, \ldots, Q_l\}], \mathbf{k}, \mathbf{p}', R$)
13:     return $\{Q_1, Q_2, \ldots, Q_l\}$
14: procedure KPTreeDFS($\mathcal{M}, \mathbf{k}, R$)
15:     $\{Q_1, Q_2, \ldots, Q_l\} \leftarrow$ PTreeDFS($\mathcal{M}, \mathbf{k}, \mathbf{0}^{l-1}, R$)
16:     if $Q_l \neq \emptyset$ then
17:         for $i \leftarrow l, l-1, \ldots, l - end0(\mathbf{k})$ do
18:             if $i > 0$ and $\mathbf{k}[i] < \kappa(G_i)$ then
19:                 $\mathbf{k}' \leftarrow \mathbf{k}$
20:                 $\mathbf{k}'[i] \leftarrow \mathbf{k}'[i] + 1$
21:                 KPTreeDFS($\mathcal{M}[\{Q_1, Q_2, \ldots, Q_l\}], \mathbf{k}', R$)
22: procedure ToFrac($\mathbf{p}$)
23:     return $(F_1[\mathbf{p}[1]], F_2[\mathbf{p}[2]], \ldots, F_{l-1}[\mathbf{p}[l-1]])$    ▷ Convert $\mathbf{p}$ to its fractional form
```

Theorem 5.5. *Algorithm 2 performs generalized core decomposition, returning all nonempty gCores in* $\mathcal{M}$*.*

The time complexity of Algorithm 2 is $O(\prod_{i=1}^{l} \kappa(G_i) \cdot \prod_{i=1}^{l-2} |F_i| \cdot (|\mathcal{M}| + l|V_l|) + l \cdot \prod_{i=1}^{l} \kappa(G_i) \cdot \prod_{i=1}^{l-1} |F_i| + \sum_{i=1}^{l-1} d_i^2 \log d_i)$. The space complexity of Algorithm 2 is $O(l \cdot (V_{max} + \sum_{i=1}^{l-1} |F_i|))$.

6 GCORE INDEXING (GCI)

In this section, we study how to store and index the results of the GCD problem to enable fast search of any $(\mathbf{k}, \mathbf{p})$-core.

6.1 Storage and Index Structure

When GCD+ (Algorithm 2) terminates, it conceptually generates a subtree of the KP-tree where a $\mathbf{p}$-node nested in a $\mathbf{k}$-node uniquely represents a nonempty $(\mathbf{k}, \mathbf{p})$-core. By materializing this subtree of KP-tree, a storage and index structure can be designed.

Specifically, for each $\mathbf{k}$-node in it, we use a hash table to map $\mathbf{k}$ to the $\mathbf{k}$-node to support fast locating the node. To store all nonempty gCores in the P-tree nested in the $\mathbf{k}$-node, say T, we augment the structure of T in the following way:

(1) For any leaf node N in T, we add a dummy node L to represent an empty gCore and designate N as the parent of L.
(2) Let N and N' be two nodes in T representing gCores Q and Q', respectively, and N' is the *leftmost* child of N. We store the set $Q - Q'$ along with the edge between N and N'. N' is said to be the leftmost child of N if $N' \prec N''$ for all siblings N'' of N', where $\prec$ is a total order of N's children as given below.

Definition 6.1. Let N' and N'' be two children of a node N. We have $N' \prec N''$ if $end0(\mathbf{p}') < end0(\mathbf{p}'')$, where $\mathbf{p}'$ and $\mathbf{p}''$ are the vectors associated with N' and N'', respectively.

Figure 5 (a) illustrates an augmented P-tree. Theorem 6.2 ensures that the augmented P-tree is a compact storage of all gCores corresponding to the nodes in the P-tree. Therefore, in the rest of the paper, when we mention P-tree/KP-tree, we are referring to the constructed storage and index structure as described above.

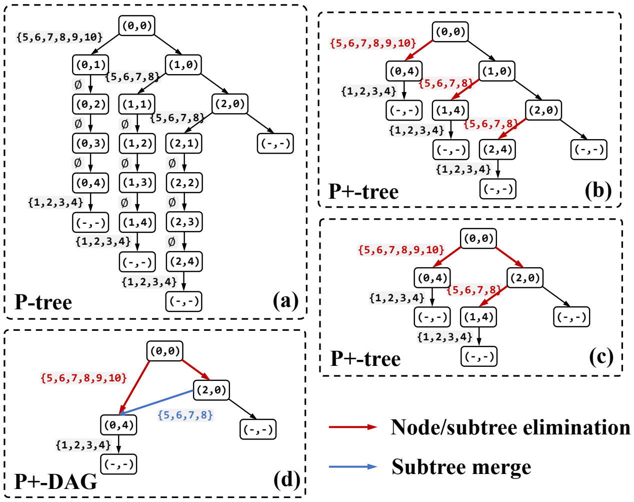


Figure 5: Augmented structures nested in the k-node of the KP-tree index of the example multi-layer graph in Figure 3, where k $= (3, 3, 3)$. (a) The original augmented P-tree. (b)-(d) The compact structures obtained by applying different compaction techniques.

Algorithm 3 GCS+ (Index-based gCore Search)

```
Input: The KP-tree for an $l$-layer graph $\mathcal{M}$, a vector $\mathbf{k} \in \mathbb{N}^l$ and a vector $\mathbf{p} \in [0,1]^{l-1}$
Output: The $(\mathbf{k},\mathbf{p})$-core in $\mathcal{M}$
 1: Find the $\mathbf{k}$-node from the hash table of the KP-tree with the key $\mathbf{k}$
 2: $T \leftarrow$ the P-tree nested in the $\mathbf{k}$-node
 3: $N \leftarrow$ Search$(T, \mathbf{p})$
 4: return Recover$(N)$
 5: procedure Search$(T, \mathbf{p})$
 6:     $N \leftarrow$ the root of the P-tree $T$
 7:     $i \leftarrow 1$
 8:     while $i < l$ do
 9:         $(N, i) \leftarrow$ Forward$(N, i, \mathbf{p})$
10:     return $N$
11: procedure Forward$(N, i, \mathbf{p})$
12:     if $\mathbf{p}'[i] < \mathbf{p}[i]$ then        ▷ $\mathbf{p}'$ is the vector (in the fractional form) associated with $N$
13:         for each child $N'$ of $N$ do
14:             if the vectors associated with $N'$ and $N$ are different in their $i$-th element then
15:                 return $(N', i)$
16:     else
17:         return $(N, i+1)$
18: procedure Recover$(N)$
19:     $Q \leftarrow \emptyset$
20:     while $N$ is not a dummy leaf do
21:         $N' \leftarrow$ the leftmost child of $N$
22:         $S \leftarrow$ the set of vertices associated with the edge between $N$ and $N'$ in the P-tree
23:         $Q \leftarrow Q \cup S$
24:         $N \leftarrow N'$
25:     return $Q$
```

Theorem 6.2. *Let N be the node representing the gCore Q. We have that Q equals the union of vertex sets associated with the edges on the leftmost path from N to a leaf node, where the leftmost path of N is obtained by following the leftmost child of N, the leftmost child of the leftmost child of N ... until a leaf node is reached.*

6.2 Index-based GCS Algorithm

The KP-tree index proposed in Section 6.1 enables a fast two-phase gCore search (GCS) algorithm (Algorithm 3). Given vectors $\mathbf{k} \in \mathbb{N}^l$ and $\mathbf{p} \in [0,1]^{l-1}$, it retrieves the $(\mathbf{k},\mathbf{p})$-core working as follows :

(1) Find a node N in the P-tree nested in the $\mathbf{k}$-node of the KP-tree that represents the $(\mathbf{k},\mathbf{p})$-core.
(2) Compute the union of all vertex sets associated with the edges on the leftmost path of N (Theorem 6.2).

The $\mathbf{k}$-node can be easily found from the hash table of the KP-tree with the key $\mathbf{k}$ (line 1). Since $\mathbf{p}$ is a real vector, it is very likely that $\mathbf{p} \notin F_1 \times F_2 \times \cdots \times F_{l-1}$, and therefore the $\mathbf{p}$-node is not indexed in the KP-tree. To handle this, we find a $\hat{\mathbf{p}}$-node in the P-tree such that $(\mathbf{k},\mathbf{p})$-core is identical to the $(\mathbf{k},\hat{\mathbf{p}})$-core. The proof of Lemma 5.1 (Appendix M of [24]) provides us a method to efficiently identify $\hat{\mathbf{p}}$: For $i = 1, 2, \ldots, l-1$, $\hat{\mathbf{p}}[i]$ is the smallest element in F_i that is not less than $\mathbf{p}[i]$. If $\mathbf{p} \in F_1 \times F_2 \times \cdots \times F_{l-1}$, we have $\hat{\mathbf{p}} = \mathbf{p}$.

After knowing $\hat{\mathbf{p}}$, Algorithm 3 calls Procedure Search (lines 5–10) in line 3 to search for the $\hat{\mathbf{p}}$-node. Then, Procedure Recover (lines 18–25) is called in line 4 to get the union of vertex sets on the leftmost path of the $\hat{\mathbf{p}}$-node, and the result is subsequently returned. The pseudocode for Procedure Search and Procedure Recover are clear, and we leave the detailed description in Appendix E of [24].

Theorem 6.3. *Algorithm 3 returns the $(\mathbf{k},\mathbf{p})$-core in $\mathcal{M}$.*

Algorithm 3 runs in $O(\sum_{i=1}^{l-1} |F_i| + |Q|)$ time and $O(|Q|)$ space, where $|Q|$ is the size of the target $(\mathbf{k},\mathbf{p})$-core.

7 COMPACTION OF P-TREES

The gCores represented by nodes in a P-tree may be not distinct. For example, in Figure 5 (a), the $\mathbf{p}$-nodes for $\mathbf{p} = (2,1)$, $(2,2)$, $(2,3)$, and $(2,4)$ all represent the gCore $\{1,2,3,4\}$. Storing these nodes incurs unnecessary storage overheads and computational costs. In this section, we explore methods to eliminate unnecessary nodes from a P-tree while ensuring that all gCores can still be correctly retrieved using Algorithm 3 on the resulting compact structure.

7.1 Foundations

We begin by introducing the foundations of our P-tree compaction methods. In a P-tree, two nodes N and N' are considered *redundant*, denoted as $N \cong N'$, if they represent the same gCore. The binary relation $\cong$ is reflexive, symmetric, and transitive, making it an equivalence relation. The equivalence class of a node N under $\cong$ is denoted as $[N] = \{N' | N' \cong N\}$.

Definition 7.1. The vector $\mathbf{p}$ associated with a node $N \in [N]$ is *maximal* if $\mathbf{p}' \leq \mathbf{p}$ for all vectors $\mathbf{p}'$ associated with nodes $N' \in [N]$.

Theorem 7.2. *The maximal vector for $[N]$ is unique.*

The following theorem helps to get the maximal vector for $[N]$.

Theorem 7.3. *Let Q be the $(\mathbf{k},\mathbf{p})$-core represented by N in the P-tree and let $\hat{\mathbf{p}}$ be the maximal vector for $[N]$. We have $\hat{\mathbf{p}}[i] = \min_{v \in Q} \phi(v, Q_i)$ for $i = 1, 2, \ldots, l-1$, where Q_i is the $\mathbf{k}[i]$-core of $G_i[Q]$ and $\phi(v, Q_i)$ is defined in Definition 3.2.*

The maximal vector for $[N]$ can uniquely characterize $[N]$:

Lemma 7.4. *For two nodes N and N' in a P-tree, we have $N \cong N'$ if and only if the maximal vectors for $[N]$ and $[N']$ are identical.*

7.2 Node Elimination

For any two nodes N and N' in a P-tree, if $N \cong N'$ and N' is N's only child, we can remove N because the gCore represented by N can be identically obtained by searching for the gCore represented by N'. If N has a parent, we link N' to N's parent as a new child. We call the process "node elimination".

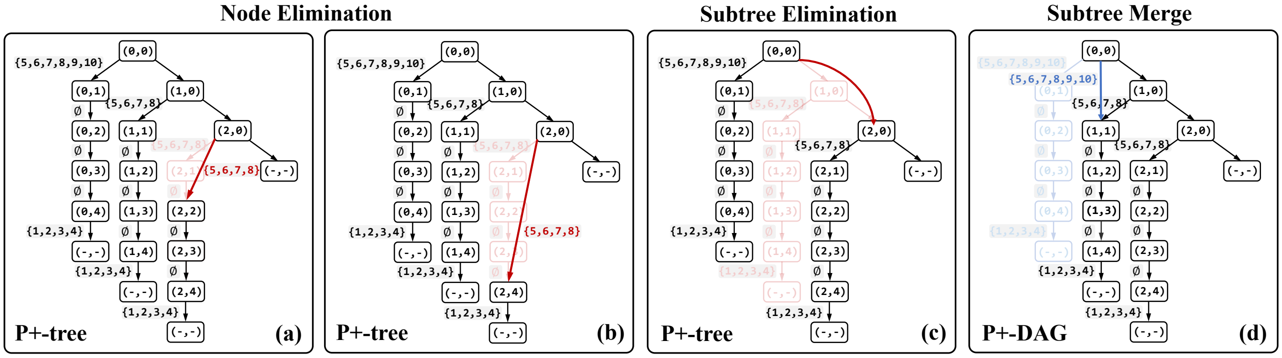


Figure 6: Illustration of P-tree compaction techniques: (a) and (b) node elimination, (c) subtree elimination, and (d) subtree merge.

Example. An example of node elimination is shown in Figure 6 (a). Let N_0, N_1 and N_2 be the nodes with $\mathbf{p} = (2,0)$, $(2,1)$ and $(2,2)$, respectively. We have $N_1 \cong N_2$ and N_2 is N_1's only child. Therefore, N_1 can be removed. After removing N_1, N_0 becomes N_2's parent.

After applying node elimination, the obtained tree is no longer a P-tree as the vector associated with N' is not an immediate suffix successor (Definition 5.2) of the new parent's vector. We name the new tree *P+-tree*, which satisfies the following property.

Property 4. *Let N and N' be two nodes in a P+-tree, and let $\mathbf{p}$ and $\mathbf{p}'$ be the vectors associated with N and N', respectively. If N is the parent of N', $\mathbf{p}'$ is a suffix successor of $\mathbf{p}$, that is: (1) $\mathbf{p}$ and $\mathbf{p}'$ are different in exactly one element, say $\mathbf{p}[i]$ and $\mathbf{p}'[i]$; (2) $\mathbf{p}[i] < \mathbf{p}'[i]$; and (3) $\mathbf{p}[j] = \mathbf{p}'[j] = 0$ for all $j > i$.*

The node elimination procedure presented above can be repeatedly applied to a P+-tree until no node can be removed anymore. The following theorem guarantees the correctness of applying Algorithm 3 on a P+-tree to solve the GCS problem.

Theorem 7.5. *Given a P-tree and the P+-tree obtained by repeatedly applying the node elimination procedure, when the $(\mathbf{k},\mathbf{p})$-core represented by a node N in the P-tree is queried, Procedure Search in Algorithm 3 returns a node N^* in the P+-tree. We have (1) $N \cong N^*$; (2) The sets (except $\emptyset$) on the leftmost path from N' to a leaf node in the P-tree are all retained on the leftmost path from N^* to a leaf node in the P+-tree, where N' is the copy of N^* in the P-tree.*

Example. The P+-tree shown in Figure 6 (b) is obtained by eliminating the nodes with vectors $\mathbf{p} = (2,1), (2,2)$ and $(2,3)$. When the $(\mathbf{k},\mathbf{p})$-core for $\mathbf{p} = (2,2)$ is queried, Procedure Search in Algorithm 3 returns the $\mathbf{p}^*$-node N^*, where $\mathbf{p}^* = (2,4)$. The vertex set $\{1,2,3,4\}$ on the leftmost path of N^* is returned as the $(\mathbf{k},\mathbf{p})$-core.

Redundant Node Detection. Lemma 7.4 implies an easier way to test the equivalence between two nodes N and N' in a P-tree if N' is N's only child as described in the following theorem.

Theorem 7.6. *Let N and N' be two nodes in a P-tree, and N' is N's only child. Let $\hat{\mathbf{p}}$ be the maximal vector for $[N]$ and $\mathbf{p}'$ be the vector associated with N'. We have $N \cong N'$ if and only if $\mathbf{p}' \le \hat{\mathbf{p}}$.*

7.3 Subtree Elimination

If a node N in a P-tree has more than one child, node elimination can never be applied to N. To overcome this limitation, we extend node elimination to subtree elimination in this subsection.

First, let us define some concepts. For a node N in a P-tree, based on the order $\prec$ (Definition 6.1) on N's children, we have a node N' is the rightmost child of N if $N'' \prec N'$ for all siblings N'' of N'. $N_1, N_2, \cdots, N_m$ is the rightmost path of N if $N_1 = N$, N_{i+1} is the rightmost child of N_i for $1 \le i < m$ and N_m is a leaf node.

Definition 7.7. For a non-leaf node N in a P-tree, let N' be the rightmost child of N. The subtree rooted at N except the subtree rooted at N' is called the preceding subtree rooted at N.

Definition 7.8. Let T and T' be two subtrees in a P-tree rooted at R and R', respectively. T and T' are redundant if

(1) $R \cong R'$, that is, R and R' are redundant;
(2) R and R' have the same number of children, and for the i-th child N_i of R and the i-th child N'_i of R', the subtrees rooted at N_i and N'_i are redundant.

In other words, T is isomorphic to T' in terms of the node redundancy relationship $\cong$, which we denote by $T \cong T'$.

If $T \cong T'$, T and T' have the same number of nodes, and there exists a bijection f from the nodes of T to the nodes of T':

$$f(N) = \begin{cases} R' & \text{if } N = R, \\ \text{the } i\text{-th child of } f(N') & \text{if } N \text{ is the } i\text{-th child of its parent } N' \end{cases} \tag{2}$$

Under the bijection f, we have

(1) $N \cong f(N)$ for all nodes N in T;
(2) N is the parent of N' in T if and only if $f(N)$ is the parent of $f(N')$ in T'.

The fact that every node N in T is redundant w.r.t. $f(N)$ in T' makes T a redundant subtree. The following *subtree elimination* procedure removes redundant preceding subtrees from a P-tree.

For two nodes N and N' in a P-tree, and N' is N's rightmost child, if the preceding subtree rooted at N, say T, is isomorphic to the preceding subtree rooted at N', say T', we remove T from the P-tree. If N has a parent, we link N' to N's parent as a new child. An example of subtree elimination is illustrated in Figure 6 (c).

Clearly, node elimination presented in Section 7.2 is a special case of subtree elimination when N' is N's only child. After subtree elimination, the obtained tree is also a P+-tree. Besides, subtree elimination can also be repeatedly applied to a P+-tree. Theorem 7.5 still holds if "node elimination" is replaced by "subtree elimination".

Redundant Subtree Detection. By extending the redundant node detection method described in Theorem 7.6, we provide an efficient way to identify redundant subtrees in a P-tree. The basic idea is to define a unique signature for a subtree (Definition 7.9) and test if two subtrees are isomorphic by comparing their signatures using the following Theorem 7.10.

Definition 7.9. Let T be a subtree in a P-tree. The signature of T is defined as the element-wise minimum of the maximal vectors of $[N]$ for all nodes N in T.

Theorem 7.10. *Let N and N' be two nodes in a P-tree with vectors $\mathbf{p}$ and $\mathbf{p}'$, respectively, and N' is a descendant of N on N's rightmost path. Let T and T' be the preceding subtrees rooted at N and N', respectively. We have $T \cong T'$ if and only if $\mathbf{p}'[i] \leq \hat{\mathbf{p}}[i]$, where $\hat{\mathbf{p}}$ is the signature of T, and i is the dimension at which $\mathbf{p}$ differs from $\mathbf{p}'$.*

7.4 Subtree Transplant

The DFS-order generation of P-tree nodes (Algorithm 2) guarantees that for any two nodes N and N' in a P-tree, where N' is the rightmost child of N, the preceding subtree rooted at N, say T, is generated prior to N' and the preceding subtree T' rooted at N'. When node N' is generated, we can determine whether the subtrees T and T' are isomorphic using Theorem 7.10. It is important to note that T' has not been generated at this point. If we find that $T \cong T'$, we can reuse the structure of T to directly obtain T' without removing T and generating T' from scratch. This approach is referred to as "*subtree transplant*". Due to space limitations, the implementation details are given in Appendix F of [24].

7.5 Subtree Merge

The condition for subtree elimination is strict. For any preceding subtrees T and T' rooted at N and N', respectively, subtree elimination cannot be applied to T if N' is not N's rightmost child or T and T' are not isomorphic. The latter case is more likely to happen if N has more children. In this subsection, a *subtree merge* approach is proposed to further reduce the redundancy between T and T'.

The idea is to conceptually divide T and T' into smaller substructures, called "branches", and in which, seek redundant subtrees that can be eliminated. The concept of branch is defined as follows:

Definition 7.11. Given two nodes N and N' in a P-tree and N' is a child of N, the subtree formed by N and the subtree rooted at N' is called a branch, denoted by B_i^N, where i is the dimension at which the vectors associated with N and N' differ.

The *subtree merge* approach works as follows. Let N and R be two nodes in a P-tree, and N is a child of R. We consider each pair of branches B_i^R and B_i^N within preceding subtrees of R and N, respectively. If there exist nodes R_1 and N_1 on the rightmost path of R in B_i^R and N in B_i^N, respectively, such that the subtrees rooted at them, say T and T', are isomorphic, we merge T into T'. This can be done by removing T and making N_1 a new child of R_1's father.
Example. Figure 6 (d) illustrates an example of subtree merge. Let R, R_1, N, N_1 be the nodes with vectors $\mathbf{p} = (0, 0), (0, 1), (1, 0), (1, 1)$, respectively. The subtrees rooted at R_1 and N_1 are denoted as T and T', respectively. R and T form the branch B_2^R, while N and T' form the branch B_2^N. In this example, B_2^R and B_2^N are exactly the preceding subtrees rooted at R and N, respectively. Although $B_2^R \ncong B_2^N$, the subtrees T and T' within them are isomorphic. Therefore, we merge T into T' by removing T and making R the parent of N_1.

Applying subtree merge, as shown in the example above, leads to a node having more than one parent, so the obtained data structure is no longer a tree but a directed acyclic graph (DAG). We call it P+-DAG. Obviously, subtree merge can be repeatedly applied to a P+-DAG until no subtrees can be merged.

Notably, subtree elimination and subtree merge complement each other. By applying subtree merge to a P+-tree obtained by applying the subtree elimination procedure, we can get a more succinct storage of all gCores. The following theorem ensures Algorithm 3 can be applied to solve the GCS problem on a P+-DAG.

Theorem 7.12. *Given a P+-tree and the P+-DAG obtained by repeatedly applying the subtree merge procedure, when the gCore represented by a node N in the P+-tree is queried, Procedure* Search *in Algorithm 3 returns a node N^* in the P+-DAG. We have (1) $N \cong N^*$; (2) The sets (except $\emptyset$) on the leftmost path from N' to a leaf node in the P+-tree are all retained on the leftmost path from N^* to a leaf node in the P+-DAG, where N' is the copy of N^* in the P+-tree.*

Redundant Subtree Detection. Based on the signatures of subtrees (Definition 7.9), we present a fast method to detect redundant subtrees within two branches in Appendix G.1 of [24]. In [24], we also provide a pivot-based redundant subtree detection algorithm (Appendix G.2) followed by a subtree-transplant-based implementation of subtree merge (Appendix G.3). This approach enables us to apply subtree merge to any pair of branches in $O(m + n)$ time, where m and n are the lengths of the rightmost paths in the two branches.

Putting It All Together. Consider the P-tree shown in Figure 5 (a), which consists of 19 nodes and 15 edges associated with vertex sets. Figure 5 (b), (c) and (d) show the P+-tree/P+DAG obtained by applying the proposed compaction techniques. Through repeated node elimination, 10 nodes and 6 edges associated with vertex sets are retained in the P+-tree, as shown in Figure 5 (b). Subsequent applications of subtree elimination further reduce the size of the P+-tree, which has 5 nodes and 4 edges with sets (see Figure 5 (c)). By continuing applying the subtree merge, we finally have a P+-DAG in Figure 5 (d), in which only 5 nodes and 3 sets are retained.

8 EXPERIMENTS

8.1 Experimental Setup

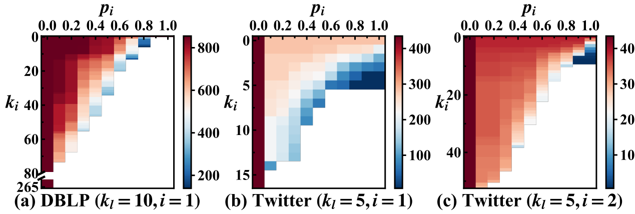
We implemented all algorithms in C++, except for the (k, Ψ)-NMC search algorithm, which was obtained from [18]. All experiments were performed on a server with an Intel Xeon Gold 5218R processor and 754GB of RAM, running 64-bit Ubuntu 22.04.
Datasets. We select 3 real-world pillar multi-layer graphs and construct 7 GMGs using publicly available datasets. The statistics of these graphs are presented in Table 1, where $|\mathcal{V}|$ denotes the total number of distinct vertices among all layers and $|E(C)|$ counts cross-layer edges with one endpoint on the layer of users' interest. The details of these graphs are given in Appendix H of [24].
Algorithms. To the best of our knowledge, no existing cohesive subgraph models can handle GMGs. Therefore, we compare our gCore model with the k-core model [2], the multi-layer core (d-CC) model [41] and the relational community model [16] designed for

[1] https://www.twitter.com/

[2] https://www.themoviedb.org/

Table 1: Properties of graphs used in experiments.

Graph	$\lvert\mathcal{V}\rvert$	$\lvert E(\mathcal{G})\rvert$	$\lvert E(C)\rvert$	#Vertex Types	l
SacchCere (SC) [12]	6750	247,152	39,420	1	7
ObamaInIsrael (Oii) [28]	2,279,535	3,827,964	4,559,070	1	3
Friendfeed (FF)[9]	505,104	18,673,521	1,010,208	1	3
6-NG [27]	4,500	15,787	24,001	5	5
9-NG [27]	6,750	24,264	36,015	5	5
DBLP [33]	41,892	280,707	381,176	2	2
Twitter[1]	47,280	445,287	89,775	3	3
Movie[2]	251,742	1,183,167	502,821	2	4
Aminer-5 [35]	2,890,443	14,536,094	7,730,034	3	5
Aminer-10 [35]	4,650,693	118,763,984	14,384,941	3	5



(a) DBLP ($k_l = 10, i = 1$) **(b) Twitter ($k_l = 5, i = 1$)** **(c) Twitter ($k_l = 5, i = 2$)**

Figure 7: Size matrices of gCores on DBLP and Twitter.

simple graphs, MPNs, and HINs, respectively. Besides, the meta-path-based cohesive subgraph model (k, Ψ)-NMC [18] proposed for HINs is also compared in the effectiveness evaluation. Specifically, we compare the following algorithms:

- KC: k-core computation algorithm in [2]. By default, we run KC only on the layer of users' interest.
- DCC: d-CC computation algorithm in [41]. In our implementation, we extend DCC to compute the multi-layer $\mathbf{k}$-core.
- RCD: Relational community detection algorithm in [16]. We adapt the algorithm to GMGs by designating vertices in each layer G_i a type t_i and using constraints (t_i, t_i, k_i) for $1 \le i \le l$ and $(t_l, t_i, 1)$ for $1 \le i < l$ as the community schema. The result is denoted by $\mathbf{k}$-rc, where $\mathbf{k} = (k_1, k_2, \cdots, k_l)$.
- CSSH: The (k, Ψ)-NMC search algorithm in [18].
- GCS: Our gCore search algorithm (Algorithm 1).
- GCS+: Our KP-tree-based gCore search algorithm (Algorithm 3).
- TN: GCD+ (Algorithm 2) with KP-tree constructed (Section 6.1).
- TE: TN with node and subtree elimination (Section 7.2–7.4).
- TM: TN with subtree merge (Section 7.5).
- TEM: TN with both subtree elimination and subtree merge.

We do not evaluate the naïve GCD algorithm proposed in Section 5.1 because it is slow and cannot generate a KP-tree index to support gCore search.

8.2 Effectiveness Evaluation

8.2.1 gCore Sizes. To demonstrate how the cohesiveness constraint imposed on different layers affects gCores, we examine the size of the $(\mathbf{k}, \mathbf{p})$-core w.r.t. k_i and p_i for $1 \le i < l$. Figure 7 visualizes the size matrices obtained on DBLP and Twitter with rows and columns representing k_i and p_i, respectively, and each cell (x, y) representing the size of the $(\mathbf{k}, \mathbf{p})$-core with $k_i = x$ and $p_i = y$. We fix $k_l = 10$ for DBLP and 5 for Twitter.

We can observe that the size of the $(\mathbf{k}, \mathbf{p})$-core is always monotonically decreasing when either k_i or p_i becomes larger. It's consistent with the containment relationships among gCores given in Properties 2 and 3. Notice that all cells in the first column correspond to $p_i = 0$, meaning no constraint is imposed on the neighbor coverage fraction and thereby the cohesiveness of the vertices shown on G_i. In this case, the $(\mathbf{k}, \mathbf{p})$-core is exactly the k_l-core. A significant drop in size exhibits when increasing p_i from 0 to 0.1. This is because the k_l-core contains massive vertices with seldom, even no, cross-layer neighbors. By properly setting p_i, the $(\mathbf{k}, \mathbf{p})$-core can well filter out these vertices. Besides, a trade-off between higher intra-layer cohesiveness and higher neighbor coverage fraction, i.e., a larger k_i and a larger p_i, can be seen. Each boundary cell corresponding to the last nonempty $(\mathbf{k}, \mathbf{p})$-core as k_i or p_i increases exhibits the highest cohesiveness w.r.t. G_i for different k_i/p_i trade-offs.

8.2.2 Closeness. We compare the closeness of vertices in the k-core, $\mathbf{k}$-rc, (k, Ψ)-NMC, and $(\mathbf{k}, \mathbf{p})$-core exhibited on different layers. To measure this, we introduce two metrics:

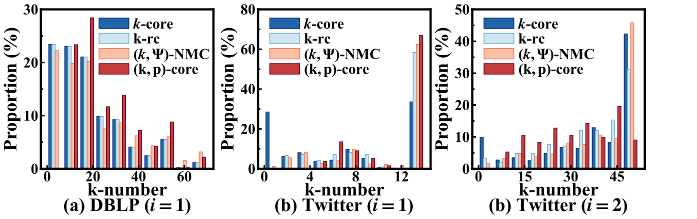
(1) **k-number**: given a vertex subset $Q \subseteq V_l$ and a real number $p^* \in [0, 1]$, the k-number of a vertex $v \in Q$ w.r.t. G_i is defined as the P-th percentile of the coreness of vertices in $N_i(v)$ within $G_i[Q]$, where $P = (1 - p^*) \times 100$. Here, the coreness of a vertex is the largest k such that there is a nonempty k-core containing the vertex in the graph. If a fraction p^* of v's cross-layer neighbors in G_i can capture enough information about v's neighborhood in G_i, a large k-number of v indicates that v's neighbors strongly engage in the community formed by neighbors of other vertices in Q, implying the close interaction between v and other vertices in Q w.r.t. G_i.

(2) **p-number**: given a vertex subset $Q \subseteq V_l$ and an integer k^*, the p-number of a vertex $v \in Q$ w.r.t. G_i is defined as the neighbor coverage fraction of v within the k^*-core on $G_i[Q]$. Assume the k^*-core is enough to characterize a desired cohesiveness in G_i, when the p-number of v is large, massive v's neighbors cohesively interact with neighbors of other vertices in Q within G_i, which also implies v's close interaction with other vertices in Q w.r.t. G_i.

We compute the k-number and p-number for vertices in the k-core, $\mathbf{k}$-rc, (k, Ψ)-NMC, and $(\mathbf{k}, \mathbf{p})$-core obtained on DBLP and Twitter. For DBLP, we set $k = 10$, $\mathbf{k} = (10, 10)$ and $\mathbf{p} = (0.7)$; and for Twitter, we set $k = 5$, $\mathbf{k} = (5, 5, 5)$ and $\mathbf{p} = (0.5, 0.5)$. When computing the k-number (resp. p-number) w.r.t G_i, we set $p^* = \mathbf{p}[i]$ (resp. $k^* = \mathbf{k}[i]$). The approach to adapt the (k, Ψ)-NMC model to GMGs is given in Appendix I of the full paper [24], where a more systematic comparison between the $(\mathbf{k}, \mathbf{p})$-core, (k, Ψ)-NMC and different versions of the $(k, \mathcal{P})$-core [11] can also be found.

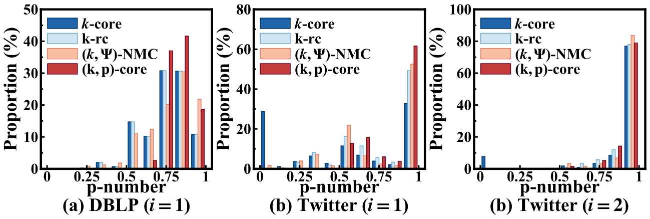
The distributions of the k-numbers are given in Figure 8. We can observe that the k-core contains massive vertices with small k-numbers, especially on DBLP and Twitter $(i = 1)$, where vertices with k-number smaller than 6 and 4 account for 23% and 37% of all vertices, respectively. The $\mathbf{k}$-rc and (k, Ψ)-NMC also have vertices with small k-numbers, accounting for 21% and 7% on DBLP and Twitter $(i = 1)$, respectively. These vertices show weak interactions with other vertices w.r.t. G_i, which may be broken by only removing several edges in G_i. However, all vertices in the $(\mathbf{k}, \mathbf{p})$-core have a relatively large k-number (no less than 10 for DBLP and 5 for Twitter). Notice that the other three models may contain a larger proportion of vertices with large k-number than the $(\mathbf{k}, \mathbf{p})$-core (see Figure 8(c)). This is because the massive vertices with small k-number enrich the subgraph induced by their cross-layer neighbors in G_i, and thereby many vertices have their coreness increased.

Figure 9 shows the results on p-numbers, which exhibit similar distributions to the k-numbers. As DBLP is very dense in G_0 (a term



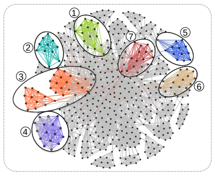
(a) DBLP ($i = 1$) (b) Twitter ($i = 1$) (b) Twitter ($i = 2$)

Figure 8: Distribution of the k-numbers of vertices in the k-core, the k-rc, the (k, Ψ)-NMC and the (k, p)-core on DBLP and Twitter.



(a) DBLP ($i = 1$) (b) Twitter ($i = 1$) (b) Twitter ($i = 2$)

Figure 9: Distribution of the p-numbers of vertices in the k-core, the k-rc, the (k, Ψ)-NMC and the (k, p)-core on DBLP and Twitter.



Model	No.	Size	Top-5 Terms
k-core	-	566	**data, system, database, query, databases**
(k, p)-core	①	11	demonstration, repository, xml, active, views
	②	11	application, design, technology, oracle, replication
	③	26	manager, performance, storage, database, memory
	④	11	system, incomplete, advanced, inconsistent, infomix
	⑤	11	hosting, sql, server, runtime, microsoft
	⑥	11	distributed, continuous, server, media, database
	⑦	12	distributed, stream, engine, operation, borealis

Figure 10: Visualization (left) and terms (right) of the largest connected component of the 10-core on DBLP and all connected components of the (k, p)-core within it, where $\mathrm{k} = (10, 10)$ and $\mathrm{p} = (0.757)$.

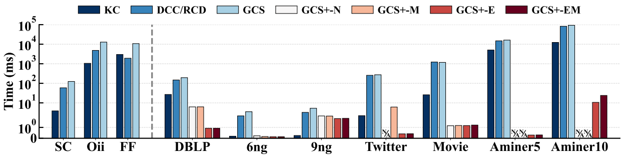


Figure 11: Runtime of cohesive subgraph search algorithms.

similarity layer), only a small proportion of vertices in the k-core, the $\mathbf{k}$-rc and (k, Ψ)-NMC have small p-numbers (about 3% vertices with p-number less than 0.5). The k-core on Twitter has 28% and 7% vertices with p-number equal to 0 for $i = 1$ and 2, respectively. These vertices have no interactions with other vertices w.r.t. G_i, making the k-core show very sparse connections in terms of G_i. It is consistent with the significant $(\mathbf{k}, \mathbf{p})$-core size drop we observed in Section 8.2.1 when increasing p_i from 0 to 0.1. The $\mathbf{k}$-rc and (k, Ψ)-NMC have a smaller proportion of vertices with small p-numbers than the k-core, while all vertices in the $(\mathbf{k}, \mathbf{p})$-core have a large p-number (no less than 0.7 for DBLP and 0.5 for Twitter).

8.2.3 Case Study on DBLP. We apply the gCore model to conduct collaboration analysis on the DBLP dataset. We set $\mathbf{k} = (10, 10)$ and $\mathbf{p} = (p)$, where p is the largest real number such that the $(\mathbf{k}, \mathbf{p})$-core is nonempty. In this case, $p = 0.757$. Figure 10 visualizes all connected components of the $(\mathbf{k}, \mathbf{p})$-core within the largest connected component of the 10-core. Each connected component corresponds to a group of cohesively collaborated authors. For each group, we compute the top-5 frequently connected terms to represent the research interests of the authors. The sizes of the groups and their corresponding terms are also presented in Figure 10.

We can observe that the connected component of the 10-core is large and includes many connected components of the $(\mathbf{k}, \mathbf{p})$-core. Moreover, the connected component of the 10-core characterizes a group of authors with research interests in data management and database systems, which are both broad fields. However, each 7 small connected component of the $(\mathbf{k}, \mathbf{p})$-core captures a group of authors collaborating on a specific research field, e.g., the connected component labeled No. ③ corresponds to an author group working in the field of database storage management. Such groups are suitable for inviting to present tutorials on certain topics.

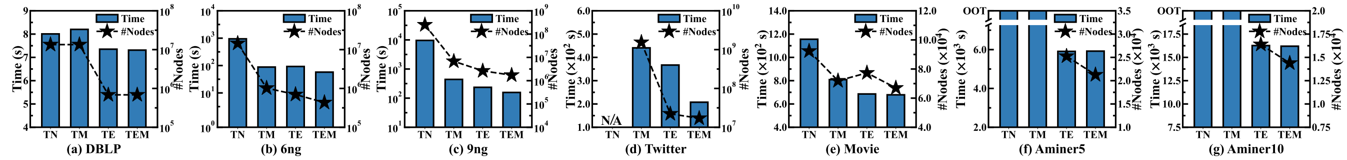
8.2.4 Case Study on Last.Fm. We apply the gCore model to discover users sharing friendships and similar musical tastes on Last.Fm, an online music sharing platform (https://www.last.fm/), and show how the result guides artist recommendation in Appendix K of [24].

8.3 Efficiency Evaluation

8.3.1 Cohesive Subgraph Search. In this experiment, we compare the efficiency of our gCore search algorithm with other core-based cohesive subgraph search algorithms on both pillar and general multi-layer graphs. On pillar multi-layer graphs, KC, DCC, and GCS are compared; and on GMGs, KC, RCD, GCS, and GCS+ are tested. The total execution time of running 100 queries is reported. Queries are generated as follows: we randomly sample 100 $(\mathbf{k}, \mathbf{p})$ pairs, and for each pair, we feed $\mathbf{k}[l]$ to KC, feed $\mathbf{k}$ to DCC and RCD, and feed $(\mathbf{k}, \mathbf{p})$ to GCS and GCS+. To avoid obtaining massive empty results, we restrict $\mathbf{k}[i] \leq \kappa(G_i)/4$ for $1 \leq i \leq l$. Besides, we distinguish GCS+ using different KP-trees. GCS+-N, GCS+-M, GCS+-E and GCS+-EM represent GCS+ searching the KP-tree generated by TN, TM, TE and TEM, respectively. Figure 11 reports the results. Empty bars with 'N/A' mean the KP-tree used by GCS+ cannot be generated within 12 hours or due to memory exceeded.

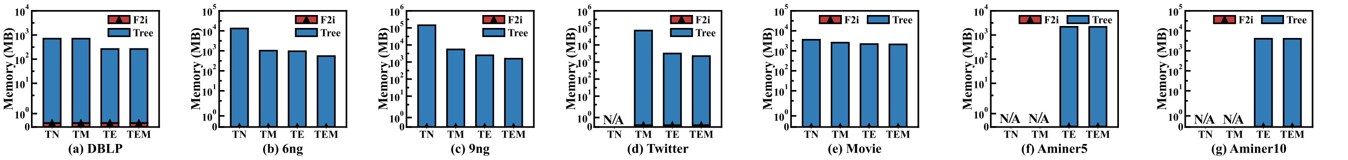
On pillar multi-layer graphs, KC uses the least time in most cases. This aligns with our expectations as KC only considers one layer. DCC runs faster than GCS because we have optimized DCC to directly access the cross-layer neighbor of a vertex by id. However, GCS is designed for GMGs, so it has to scan the adjacent list to obtain all cross-layer neighbors of a vertex.

On GMGs, KC also runs the fastest among all algorithms without using indexes. GCS attains a slightly longer runtime than RCD because it has to check whether each vertex meets the constraints imposed on the neighbor coverage fractions. With a pre-computed KP-tree, GCS+ significantly outperforms GCS by 1 − 4 orders of magnitude on the execution time on most datasets, except for 9NG where GCS+ achieves a speedup of 1.4 to 1.9 compared to GCS. We next inspect the efficiency of GCS+. In most cases, GCS+-E and GCS+-EM run faster than GCS+-N and GCS+-M. This is because applying subtree elimination contributes to a reduction in the height of P+-trees. The number of tree nodes traversed to retrieve the gCore is consequently reduced. Additionally, GCS+-EM is consistently slower than GCS+-E. This is because, in the KP-tree index, P-tree nodes are actually stored in linked blocks in the order



(a) DBLP (b) 6ng (c) 9ng (d) Twitter (e) Movie (f) Aminer5 (g) Aminer10

Figure 12: Construction time and scale of the KP-Tree.



(a) DBLP (b) 6ng (c) 9ng (d) Twitter (e) Movie (f) Aminer5 (g) Aminer10

Figure 13: Memory consumption of the KP-Tree.

they are generated. GCS+-E can leverage the cache locality during traversal on the P-tree, while GCS+-EM incurs more cache misses as subtree merge disrupts the sequential storage of the tree nodes.

8.3.2 *KP-tree Construction.* In this experiment, we evaluate our gCore decomposition (indexing) algorithm and P-tree compaction techniques. Specifically, we compare TN, TM, TE and TEM in terms of runtime, scale (number of nodes) and storage overhead of the output KP-tree index. Notice that Movie, Aminer-5 and Aminer-10 datasets have about 10^5, 10^9 and 10^{11} distinct values of $\mathbf{k}$, each of which corresponds to a P-tree. Constructing a complete KP-tree is infeasible under limited time and memory constraints. Moreover, not all values of $\mathbf{k}$ are of users' interest. Thus, we randomly sample 1000 values of $\mathbf{k}$ and compute corresponding P-trees for these 3 datasets. Each element $\mathbf{k}[i]$ of $\mathbf{k}$ is randomly chosen in $[0, \kappa(G_i)/4]$. Total runtime, scale and storage overhead are reported.

Runtime and Scale. Figure 12 reports the runtime and the scale of the generated KP-trees. We set a timeout (OOT) of 12h. Execution of TN on Twitter is aborted due to memory exceeded. We have the following observations. 1) TE outperforms TN in both runtime and scale of the output. It confirms the promising compaction ability of subtree elimination. 2) TM runs significantly faster than TN on 6ng, 9ng, Twitter and Movie, while falls behind on DBLP. This is because subtree merge only works for graphs with more than 2 layers. Additional isomorphic subtree detection operations slow down the execution on DBLP. 3) By using both subtree merge and subtree elimination, TEM always yields the smallest KP-tree, and in most cases, uses the shortest time.

Storage Overhead. Figure 13 reports the storage overhead of the generated KP-tree index, including the space for storing the KP-tree structure and the space for storing F_i and the mapping that maps each element in F_i to its index. We denote the data structure to store the second part by 'f2i' (fraction to index). We have the following observations. 1) F2i has neglected space cost, which is 3-9 orders of magnitude smaller than the KP-tree for all tested datasets. 2) Both subtree merge and subtree elimination contribute to reducing the storage overhead. By integrating both of them into TN, the KP-tree generated by TEM attains a 41% − 98% space reduction.

8.3.3 *Scalability.* We evaluate the scalability of the algorithms proposed in this paper, including GCS, GCS+, and TEM, which has exhibited the best performance in solving the GCI problem. Specifically, we test the runtime of these algorithms using different versions of the Aminer-10 dataset obtained by selecting a variable number of layers and subsets of vertices. The detailed results are presented in Appendix J of the full paper [24]. Here are several key findings: 1) GCS and GCS+ exhibit linear and sub-linear scalability with the size of the multi-layer graph, respectively, when the number of layers is fixed. 2) Introducing a new layer have both positive and negative impacts on the efficiency of GCS and GCS+. 3) The scale of the KP-tree generated by TEM grows linearly as the number of vertices on the layer of users' interest increases. 4) Introducing a new layer increases both the runtime of TEM and the scale of the generated KP-tree.

9 CONCLUSIONS AND FUTURE WORK

The generalized core (gCore) model is proposed on general multi-layer graphs (GMGs) by extending the k-core model designed for simple graphs. The gCore model has several elegant properties including uniqueness and containment hierarchy. Three related problems, gCore search (GCS), gCore decomposition (GCD) and gCore indexing (GCI), based on the gCore model are solved by designing efficient algorithms and data structures. The polynomial-time GCS algorithm is comparable to other algorithms for searching core-based cohesive subgraphs in multi-layer graphs in terms of execution time. The KP-tree structure represents the whole search space of the GCD problem and can serve as a compact storage and index (GCI) of all gCores. The KP-tree supports fast retrieval of any gCore Q in linear time in the size of Q and the height of the KP-tree. The compaction techniques including node/subtree elimination, subtree transplant and subtree merge significantly speed up the construction of GCI and reduce the storage overhead of the KP-tree. Extensive experiments show that the vertices in the gCores attain high closeness with each other, and the proposed algorithms are efficient in solving the three problems studied in the paper.

We outline two promising research directions for future work. First, developing efficient update mechanisms for the KP-tree index against frequent graph dynamics. By addressing this issue, we can further enhance the applicability of the gCore model and related algorithms in real-world scenarios. Secondly, exploring more extension schemes such as the one used in the (k, λ)-FirmCore. These extensions have the potential to provide further insights and solutions to the ML-CSM problem on GMGs.

ACKNOWLEDGMENTS

This work was partially supported by the National Natural Science Foundation of China (No. 62072138).

REFERENCES

[1] José Ignacio Alvarez-Hamelin, Luca Dall'Asta, Alain Barrat, and Alessandro Vespignani. 2005. k-core decomposition: A tool for the visualization of large scale networks. *arXiv preprint cs/0504107* (2005).

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

[3] Ali Behrouz, Farnoosh Hashemi, and Laks V. S. Lakshmanan. 2022. FirmTruss Community Search in Multilayer Networks. *Proc. VLDB Endow.* 16, 3 (2022), 505–518.

[4] Stefano Boccaletti, Ginestra Bianconi, Regino Criado, Charo I Del Genio, Jesendiús Gómez-Gardenes, Miguel Romance, Irene Sna-Nadal, Zhen Wang, and Massimiliano Zanin. 2014. The structure and dynamics of multilayer networks. *Physics reports* 544, 1 (2014), 1–122.

[5] Brigitte Boden, Stephan Günnemann, Holger Hoffmann, and Thomas Seidl. 2017. MiMAG: mining coherent subgraphs in multi-layer graphs with edge labels. *Knowledge and Information Systems* 50 (2017), 417–446.

[6] Iván Cantador, Peter Brusilovsky, and Tsvi Kuflik. 2011. 2nd Workshop on Information Heterogeneity and Fusion in Recommender Systems (HetRec 2011). In *Proceedings of the 5th ACM conference on Recommender systems* (Chicago, IL, USA) *(RecSys 2011)*. ACM, New York, NY, USA.

[7] Lijun Chang and Lu Qin. 2019. Cohesive subgraph computation over large sparse graphs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 2068–2071.

[8] Annie P Chiang and Atul J Butte. 2009. Systematic evaluation of drug–disease relationships to identify leads for novel drug uses. *Clinical Pharmacology & Therapeutics* 86, 5 (2009), 507–510.

[9] Mark E Dickison, Matteo Magnani, and Luca Rossi. 2016. *Multilayer social networks*. Cambridge University Press.

[10] Yixiang Fang, Kai Wang, Xuemin Lin, and Wenjie Zhang. 2022. *Cohesive Subgraph Search Over Large Heterogeneous Information Networks*. Springer.

[11] Yixiang Fang, Yixing Yang, Wenjie Zhang, Xuemin Lin, and Xin Cao. 2020. Effective and efficient community search over large heterogeneous information networks. *Proceedings of the VLDB Endowment* 13, 6 (2020), 854–867.

[12] Edoardo Galimberti, Francesco Bonchi, and Francesco Gullo. 2017. Core decomposition and densest subgraph in multilayer networks. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. 1807–1816.

[13] Farnoosh Hashemi, Ali Behrouz, and Laks VS Lakshmanan. 2022. FirmCore Decomposition of Multilayer Networks. In *Proceedings of the ACM Web Conference 2022*. 1589–1600.

[14] Jiafeng Hu, Reynold Cheng, Kevin Chen-Chuan Chang, Aravind Sankar, Yixiang Fang, and Brian YH Lam. 2019. Discovering maximal motif cliques in large heterogeneous information networks. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 746–757.

[15] Hongxuan Huang, Qingyuan Linghu, Fan Zhang, Dian Ouyang, and Shiyu Yang. 2021. Truss Decomposition on Multilayer Graphs. In *2021 IEEE International Conference on Big Data (Big Data)*. IEEE, 5912–5915.

[16] Xun Jian, Yue Wang, and Lei Chen. 2020. Effective and efficient relational community detection and search in large dynamic heterogeneous information networks. *Proceedings of the VLDB Endowment* 13, 10 (2020), 1723–1736.

[17] Daxin Jiang and Jian Pei. 2009. Mining frequent cross-graph quasi-cliques. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 2, 4 (2009), 1–42.

[18] Yangqin Jiang, Yixiang Fang, Chenhao Ma, Xin Cao, and Chunshan Li. 2022. Effective community search over large star-schema heterogeneous information networks. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2307–2320.

[19] Jungeun Kim and Jae-Gil Lee. 2015. Community detection in multi-layer graphs: A survey. *ACM SIGMOD Record* 44, 3 (2015), 37–48.

[20] Mikko Kivelä, Alex Arenas, Marc Barthelemy, James P Gleeson, Yamir Moreno, and Mason A Porter. 2014. Multilayer networks. *Journal of complex networks* 2, 3 (2014), 203–271.

[21] Victor E Lee, Ning Ruan, Ruoming Jin, and Charu C Aggarwal. 2010. A Survey of Algorithms for Dense Subgraph Discovery. *Managing and mining graph data* 40 (2010), 303–336.

[22] Yong Li and Pankaj Agarwal. 2009. A pathway-based view of human diseases and disease relationships. *PloS one* 4, 2 (2009), e4346.

[23] Boge Liu, Fan Zhang, Chen Zhang, Wenjie Zhang, and Xuemin Lin. 2019. Corecube: Core decomposition in multilayer graphs. In *Web Information Systems Engineering–WISE 2019: 20th International Conference, Hong Kong, China, January 19–22, 2020, Proceedings 20*. Springer, 694–710.

[24] Dandan Liu and Zhaonian Zou. 2023. *gCore: Exploring Cross-layer Cohesiveness in Multi-layer Graphs (technique report)*. https://github.com/SSSuperDan/gCore/blob/master/full_version.pdf

[25] Matteo Magnani and Luca Rossi. 2011. The ml-model for multi-layer social networks. In *2011 International conference on advances in social networks analysis and mining*. IEEE, 5–12.

[26] Hideo Matsuda, Tatsuya Ishihara, and Akihiro Hashimoto. 1999. Classifying molecular sequences using a linkage graph with their pairwise similarities. *Theoretical Computer Science* 210, 2 (1999), 305–325.

[27] Jingchao Ni, Shiyu Chang, Xiao Liu, Wei Cheng, Haifeng Chen, Dongkuan Xu, and Xiang Zhang. 2018. Co-regularized deep multi-network embedding. In *Proceedings of the 2018 world wide web conference*. 469–478.

[28] Elisa Omodei, Manlio De Domenico, and Alex Arenas. 2015. Characterizing interactions in online social networks during exceptional events. *Frontiers in Physics* 3 (2015), 59.

[29] Jian Pei, Daxin Jiang, and Aidong Zhang. 2005. On mining cross-graph quasi-cliques. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*. 228–238.

[30] Ahmet Erdem Saríyüce, Buğra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V Çatalyürek. 2013. Streaming algorithms for k-core decomposition. *Proceedings of the VLDB Endowment* 6, 6 (2013), 433–444.

[31] Stephen B Seidman. 1983. Network structure and minimum degree. *Social networks* 5, 3 (1983), 269–287.

[32] Loitongbam Gyanendro Singh, Anasua Mitra, and Sanasam Ranbir Singh. 2020. Sentiment analysis of tweets using heterogeneous multi-layer network representation and embedding. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 8932–8946.

[33] Yizhou Sun, Jiawei Han, Jing Gao, and Yintao Yu. 2009. itopicmodel: Information network-integrated topic modeling. In *2009 Ninth IEEE international conference on data mining*. IEEE, 493–502.

[34] Silpa Suthram, Joel T Dudley, Annie P Chiang, Rong Chen, Trevor J Hastie, and Atul J Butte. 2010. Network-based elucidation of human disease similarities reveals common functional modules enriched for pluripotent drug targets. *PLoS computational biology* 6, 2 (2010), e1000662.

[35] Jie Tang, Jing Zhang, Limin Yao, Juanzi Li, Li Zhang, and Zhong Su. 2008. Arnetminer: extraction and mining of academic social networks. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. 990–998.

[36] Jianyong Wang, Zhiping Zeng, and Lizhu Zhou. 2006. Clan: An algorithm for mining closed cliques from large dense graph databases. In *22nd International Conference on Data Engineering (ICDE'06)*. IEEE, 73–73.

[37] Xiaoliang Xu, Jun Liu, Yuxiang Wang, and Xiangyu Ke. 2022. Academic Expert Finding via $(k, \mathcal{P})$-Core based Embedding over Heterogeneous Graphs. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 338–351.

[38] Yixing Yang, Yixiang Fang, Xuemin Lin, and Wenjie Zhang. 2020. Effective and efficient truss computation over large heterogeneous information networks. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 901–912.

[39] Zhiping Zeng, Jianyong Wang, Lizhu Zhou, and George Karypis. 2006. Coherent closed quasi-clique discovery from large dense graph databases. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. 797–802.

[40] 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.

[41] Rong Zhu, Zhaonian Zou, and Jianzhong Li. 2018. Diversified coherent core search on multi-layer graphs. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 701–712.

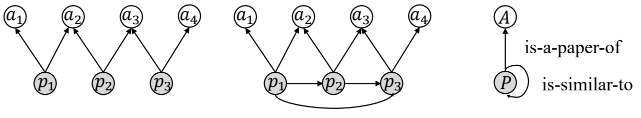
A MODEL COMPARISON

This section presents a comprehensive comparison between HINs and GMGs, as well as the cohesive subgraph models developed for HINs and the gCore model proposed in this paper.

From a static structural perspective, the GMG model studied in this paper can be equivalently translated to the HIN model, and vice versa. The translation method is provided in [19]. It should be noted that although these two models have the same representation ability, their emphasis and analytical methods differ in existing research. Specifically, the HIN model primarily focuses on the relationships between heterogeneous entities, which we say ‘heterogeneous relationships’ for short. In contrast, the GMG model incorporates the concept ‘layer’ in MPNs to highlight each relationship between homogeneous entities (‘homogeneous relationships’), which are often the most directed and informative relationships between entities. Heterogeneous relationships represented by cross-layer edges are used to integrate the analytical results on different layers. Besides, by leveraging the concept ‘layer’, the GMG model can distinguish various homogeneous relationships among a group of entities and analyze them simultaneously, which is barely considered in existing studies on HINs. From this perspective, the studied GMG model can be viewed as a more general HIN with a multi-graph-schema, where each node in the schema may have multiple self-loops representing different homogeneous relationships, while the widely studied HINs typically have a simple-graph-schema.

In the following, we will carefully present the differences and connections between the existing cohesive models on HINs and our gCore model. As described in Ref. [18], the models on HINs can be mainly classified into two types: meta-path-based models and relational-constraint-based models. We next discuss these two types of models separately.

The meta-path-based models [11, 18, 38] use meta-paths to capture the high-order relationship between entities, and based on which, apply classic cohesive subgraph models to characterize cohesively connected entities. We argue that the meta-paths used in existing models are rigid and cannot obtain a full picture of the relationships between entities to support high-quality cohesive subgraph mining or other downstream applicable tasks. The reasons are as follows: (1) The meta-paths used in existing models are always in the form of $\mathsf{A}_1 \rightarrow \mathsf{A}_2 \rightarrow \cdots \rightarrow \mathsf{A}_l$, where adjacent entities types are different. Homogeneous relationships are rarely considered when specifying meta-paths. Even worse, some works require the HIN schema to be star-shaped [18], which completely disregards the homogeneous relationships between entities. (2) Existing models on HINs are often defined based on a single user-specific meta-path. Jiang et al. have pointed out that it poses great challenges to users unfamiliar with the HIN schema or their semantic relationships to provide a meaningful meta-path. Furthermore, a meta-path only defines one particular higher-order relationship between entities, which is unlikely to comprehensively characterize their complex relationships in real-world scenarios. The problem can be alleviated by taking more meta-paths into consideration, as used in the (k, Ψ)-core model in Ref. [18]. (3) Meta-path instances fail to capture the cohesiveness of homogeneous relationships between their passing entities, which we believe has an unneglected impact on enhancing the cohesiveness among targeted entities.



(a) An HIN H_1 (b) An HIN H_2 (c) Schema

Figure 14: Limitation of meta-path-based constraint.

Let’s illustrate this with an example. Figure 14 shows two HINs, H_1 and H_2, in (a) and (b) respectively, along with their schema in (c). In both H_1 and H_2, there are four authors, a_1, a_2, a_3, and a_4, and three papers, p_1, p_2, and p_3. Consider a commonly used meta-path $\mathcal{P} = \mathsf{A} \rightarrow \mathsf{B} \rightarrow \mathsf{A}$. It can be observed that the $\mathcal{P}$-graph of H_1 and H_2 are identical. In fact, in H_2, the papers form a 3-clique, indicating their high similarity. The authors in H_2 have collaborated on three similar papers, which certainly exhibit more closeness than those in H_1 and are more likely to form a cohesive group. However, $\mathcal{P}$ fails to capture this since its meta-path instances can only reflect the connections between entities under a specific pathway pattern, rather than how the passing homogeneous entities interact with each other.

Unlike meta-path-based models, our gCore model is designed specifically for GMGs. It prioritizes the cohesiveness of each homogeneous relationship and leverages cross-layer heterogeneous relationships to enhance the cohesive groups in the target layer with the ones in other layers. The gCore model overcomes the three issues of the meta-path-based models mentioned above. Specifically, it (1) considers both homogeneous and heterogeneous relationships, (2) does not rely on user-specified meta-paths, and simultaneously considers multiple relationships between targeted entities, and (3) takes into account the enhancement of cohesiveness among non-targeted entities to the connections between targeted entities.

The relational-constraint-based model, the relational community (RC) model in [16], is defined based on a set of user-specified constraints in the form of $s = (\mathsf{E1}, \mathsf{E2}, k)$, which means “*each entity of type E1 must have at least k neighbors of type E2*”. If $\mathsf{E1} = \mathsf{E2}$, then s is a cohesive constraint imposed on homogeneous relationships, meaning that it requires the participated $\mathsf{E1}$ entities to form a k-core. Otherwise, if $\mathsf{E1} \neq \mathsf{E2}$, s puts constraints on cohesive heterogeneous relationships. It should be noticed that the RC model enables users to put constraints on multiple homogeneous and heterogeneous relationships simultaneously. From this perspective, we personally believe that the RC model is more in line with the characteristics of GMGs, and is the model that is most similar to our gCore model.

We will now highlight the differences between the RC model and our gCore model in the following three aspects: (1) Designed for HINs, the RC model is not applicable when multiple homogeneous relationships exist between entities of the same type because it’s relied on relational constraints is imposed on two or one entity types. On the opposite, our gCore model is able to distinguish and put different constraints on each different homogeneous relationship as GMGs organize entities into different ‘layers’. (2) The RC model relies on a set of user-specified relational constraints. However, as pointed out by Jiang et al. in [18], it can be challenging for users unfamiliar with the HIN schema or the semantic relationship between entities to provide meaningful relational constraints. In contrast, our gCore model takes into account each homogeneous relationship and the heterogeneous relationship between target entities and non-target entities. The heterogeneous relationships between non-target entities are ignored to simplify the model because we believe that they have less impact on enhancing the connections between the target entities. (3) When imposing constraints on heterogeneous relationships, the RC model uses an absolute threshold, while our gCore model uses a relative threshold p. This requires each entity of the target type to have at least p fraction of neighbors to cohesively connect with the neighbors of other entities. We are of the opinion that a relative threshold is more reasonable since entities may have varying numbers of cross-layer neighbors, and the same number of neighbors may characterize entities with different degrees to different levels. This approach is inspired by the (k, p)-core model [40].

B EFFICIENT IMPLEMENTATION OF ALGORITHM 1

In this section, we will present an efficient implementation of the GCS algorithm described in Algorithm 1. Let us first introduce the basic idea. For ease of notation, we use Q_l^t to denote the value of Q_l at the beginning of the t-th running of the **repeat** loop (lines 2–10) in Algorithm 1 and use Q_i^t to denote the value of Q_i obtained in line 6 of Algorithm 1 during the t-th running of the **repeat** loop.

Lemma B.1. *For $i = 1, 2, \cdots, l$, $Q_i^{t+1} \subseteq Q_i^t$ holds, where $t \in \mathbb{Z}$.*

Proof. The vertex peeling process ensures $Q_l^{t+1} \subseteq Q_l^t$. Suppose the value of Q_l^t and Q_l^{t+1} becomes C_l^t and C_l^{t+1} after line 4 in corresponding loop is executed, respectively. We have that C_l^t is the $\mathbf{k}[l]$-core of $G_l[Q_l^t]$ and C_l^{t+1} is the $\mathbf{k}[l]$-core of $G_l[Q_l^{t+1}]$. Apparently, $C_l^{t+1} \subseteq C_l^t$. According to the algorithm, Q_i^t is the $\mathbf{k}[i]$-core of $G_i[C_l^t]$, and Q_i^{t+1} is the $\mathbf{k}[i]$-core of $G_i[C_l^{t+1}]$. Obviously, we have $Q_i^{t+1} \subseteq Q_i^t$. Overall, the lemma holds. □

Based on Lemma B.1, we dynamically maintain each Q_i for $1 \le i < l-1$ via vertex peeling besides Q_l, rather than calculating it from scratch. Arrays are utilized to realize this efficiently. Specifically, for each layer G_i, we set up two arrays $vert_i$ and pos_i to keep vertices in Q_i and the position of these vertices in $vert_i$, respectively. Besides, two offsets s_i and e_i are attached to $vert_i$, dividing it into three parts. We stipulate that: (1) $vert_i[0 : s_i)$ keeps all '*discarded*' vertices that have been removed from Q_i; (2) $vert_i[s_i : e_i)$ keeps the '*inactive*' vertices to be removed from Q_i, which will become *discarded* after a batch discard operation. (3) and the last part $vert_i[e_i : |V_i|)$ keeps all '*active*' vertices that currently remain in the Q_i. As stated in Section 4, once no vertices violating Definition 3.3 can be removed from $vert_l[e_i : |V_i|)$, all remaining vertices form the $(\mathbf{k}, \mathbf{p})$-core.

Next, we define two basic operations, namely Remove and Peel. Remove converts a vertex $v \in Q_i$ from active state to inactive state. It can be realized by exchanging the positions of v and $vert_i[e_i]$ within $vert_i$ and then increasing e_i by 1. Peel iteratively identifies vertices in Q_i with degrees smaller than $\mathbf{k}[i]$ and inactivates them. Peel can be realized by iteratively checking active neighbors of each inactive vertex, updating their degree, and inactivating those whose intra-layer degree drops below $\mathbf{k}[i]$ by applying Remove. Clearly, after applying Peel, vertices remained in Q_i form a $\mathbf{k}[i]$-core. The pseudocode for Remove and Peel is given in Procedure Remove (lines 29–34) and Peel (lines 35–44) in Algorithm 4, respectively.

Algorithm 4 EffGCS (Efficient Generalized Core Search)

```
Input: an $l$ layer graph $\mathcal{M}$, $\mathbf{k} \in \mathbb{N}^l$ and $\mathbf{p} \in [0, 1]^{l-1}$.
Output: the $(\mathbf{k}, \mathbf{p})$-core in $\mathcal{M}$.
      /* Lines 1–9 perform initialization. */
 1: for $i \leftarrow 1, 2, \cdots, l$ do
 2:     $j \leftarrow 0$
 3:     for $v \in V_i$ do
 4:         $vert_i[j] \leftarrow v, pos_i[v] = j$
 5:         $j \leftarrow j + 1$
 6:     $s_i \leftarrow 0, e_i \leftarrow 0$
 7:     for $j \leftarrow 0, 1, \cdots, |V_i| - 1$ do
 8:         if $deg_i[vert_i[j]] < \boldsymbol{k}[i]$ then
 9:             Remove($vert_i[j], i$)
      /* Lines 10–27 perform vertex peeling. */
10: while $s_l \neq e_l$ do
11:     Peel($l$)
12:     for $i \leftarrow 1, 2, \cdots, l-1$ do
13:         for $j \leftarrow s_l, \cdots, e_l - 1$ do
14:             for $v \in N_i(vert_l[j])$ do
15:                 if $pos_i[v] \ge e_i$ then
16:                     $deg_{i,l}[v] \leftarrow deg_{i,l}[v] - 1$
17:                     if $deg_{i,l}[v] = 0$ then
18:                         Remove($v, i$)
19:     $s_l \leftarrow e_l$
20:     for $i \leftarrow 1, 2, \cdots, l-1$ do
21:         Peel($i$)
22:         for $v \in N_l(vert_i[j])$ do
23:             if $pos_l[v] \ge e_l$ then
24:                 $deg_{l,i}[v] \leftarrow deg_{l,i}[v] - 1$
25:                 if $deg_{l,i}[v]/|N_i(v)| < \boldsymbol{p}[i]$ then
26:                     Remove($v, l$)
27:         $s_i \leftarrow e_i$
28: return $\{vert_l[i] | e_l \le i < |V_l|\}$
29: procedure Remove($v, i$)
30:     $vert_i[pos_i[v]] = vert_i[e_i]$
31:     $pos_i[vert_i[e_i]] = pos_i[v]$
32:     $vert_i[e_i] = v$
33:     $pos_i[v] = e_i$
34:     $e_i \leftarrow e_i + 1$
35: procedure Peel($i$)
36:     $j \leftarrow s_i$
37:     while $j \neq e_i$ do
38:         $v \leftarrow vert_i[j]$
39:         for $u \in N_i(v)$ do
40:             if $pos_i[u] \ge e_i$ then
41:                 $deg_i[u] \leftarrow deg_i[u] - 1$
42:                 if $deg_i[u] < \mathbf{k}[i]$ then
43:                     Remove($u, i$)
44:         $j \leftarrow j + 1$
```

Based on the array structures and operations introduced above, we present our efficient implementation of Algorithm 1 in Algorithm 4. For efficiency consideration, we also dynamically maintain intra-layer degrees and cross-layer degrees of each vertex. Specifically, deg_i keeps the intra-layer degree of vertices in Q_i, and $deg_{i,j}$ keeps the cross-layer degree of vertices in Q_i within Q_j. Lines 1–9 perform essential initialization. For each layer G_i, $vert_i$, pos_i, s_i and e_i are set in lines 2–6. Vertices with degrees less than $\mathbf{k}[i]$ are identified as the first batch of inactive vertices, so Remove is applied to them (lines 7–9). The **while** loop in lines 10–27 performs vertex

Algorithm 5 NaïveGCD (Naïve Generalized Core Decomposition)

```
Input: An $l$-layer graph $\mathcal{M}$
Output: The collection $R$ of all $(\mathbf{k}, \mathbf{p})$-cores (and $\mathbf{k}$ and $\mathbf{p}$)
 1: $R \leftarrow \emptyset$
 2: $K \leftarrow \{\mathbf{k} | 0 \leq \mathbf{k}[i] \leq \kappa(G_i)$ for $i = 1, 2, \ldots, l\}$
 3: for $i \leftarrow 1, 2, \ldots, l-1$ do
 4:     $F_i \leftarrow$ GenFrac($\mathcal{M}, i$)
 5: $P \leftarrow \{\mathbf{p} | \mathbf{p}[i] \in F_i$ for $i = 1, 2, \ldots, l-1\}$
 6: for $\mathbf{k} \in K$ do
 7:     for $\mathbf{p} \in P$ do
 8:         $Q \leftarrow$ GCS$(\mathcal{M}, \mathbf{k}, \mathbf{p})$
 9:         $R \leftarrow R \cup \{(Q, \mathbf{k}, \mathbf{p})\}$
10: return $R$
11: procedure GenFrac($\mathcal{M}, i$)
12:     $F_i \leftarrow \emptyset$
13:     for $d \in \{deg_i(v) | v \in V_l\}$ do
14:         for $i \leftarrow 0, 1, \ldots, d$ do
15:             $F_i \leftarrow F_i \cup \{i/d\}$
16:     return $F_i$
```

peeling in each layer. In each iteration of the loop, vertices violating Constraint (1) in Definition 3.3 are iteratively removed from Q_l by applying Peel in line 11. Lines 12–18 identify and remove vertices not in $N_i[Q_l]$ from Q_i. Specifically, for each inactive vertex in layer G_l, i.e., in $vert_l[s_l, e_l)$, lines 14–16 update the cross-layer degree of its neighbors in every other layer. If a neighbor v in some Q_i has its cross-layer degree dropping to 0, v will no longer belong to Q_i, and hence Remove is applied to it (lines 17-18). After that, all inactive vertices in G_l are batch discarded (converted to discarded state) by increasing s_l to e_l (line 19). Then, for each layer G_i with $1 \leq i < l$, line 21 computes the $\mathbf{k}[i]$-core of $G_i[Q_l]$ by applying Peel. For each inactive vertex in G_i, i.e., in $vert_i[s_i, e_i]$, lines 22–24 update the cross-layer degree of its neighbors in Q_l. If it leads to the neighbor coverage fraction of a vertex $v \in Q_l$ becoming smaller than $\mathbf{p}[i]$ (violating Constraint (2) in Definition 3.3), Remove is applied to v (lines 25–26). Thereafter, all inactive vertices in G_i are batched discarded by increasing s_i to e_i (line 27). The **while** loop terminates as soon as $s_l = e_l$ holds at the end of some iteration. Finally, vertices in $vert_l[e_l : |V_l|)$ are returned as the $(\mathbf{k}, \mathbf{p})$-core in line 28.

C PSEUDOCODE FOR THE NAÏVE GCD ALGORITHM

The pseudocode for the naïve GCD algorithm introduced in Section 5.1 is given in Algorithm 5, which is self-explanatory. For each $(\mathbf{k}, \mathbf{p})$-core Q, the algorithm outputs Q as well as $\mathbf{k}$ and $\mathbf{p}$.

D SUPPLEMENTARY EXPLANATION TO GCD+ (ALGORITHM 2)

In this section, we give some supplementary explanations to the KP-tree-based GCD algorithm GCD+ (Algorithm 2) introduced in Section 5.2.2.

D.1 Description of Algorithm 2

Algorithm 2 includes two key procedures: PTreeDFS (lines 4–13) and KPTreeDFS (lines 14–21).

PTreeDFS is a recursive procedure to realize a DFS on a P-tree. Given vectors $\mathbf{k}$ and $\mathbf{p}$ as input, PTreeDFS first calls GCS (Algorithm 1) to compute the $(\mathbf{k}, \mathbf{p})$-core (line 5). Here, GCS needs to be simply adapted to return Q_l as well as $Q_1, Q_2, \ldots, Q_{l-1}$, where Q_i is the set of vertices remaining on layer G_i when GCS terminates. If $Q_l = \emptyset$, the $(\mathbf{k}, \mathbf{p}')$-cores corresponding to the descendant $\mathbf{p}'$-nodes of the $\mathbf{p}$-node are all empty according to Implication I1, so lines 7–12 can be skipped. If $Q_l \neq \emptyset$, we add $(Q_l, \mathbf{k}, \mathbf{p})$ to the result set R (line 7). Now, we completed the traversal to the $\mathbf{p}$-node in this P-tree. Next, we proceed with a DFS traversal on the descendants of the $\mathbf{p}$-node. To this end, we enumerate immediate suffix successors of $\mathbf{p}$ (lines 8–11). For each valid immediate suffix successor $\mathbf{p}'$ of $\mathbf{p}$ ($\mathbf{p}'$ is valid if $\mathbf{p}'[i] \leq |F_i|$ for all i), we recursively call PTreeDFS given $\mathcal{M}[\{Q_1, Q_2, \ldots, Q_l\}]$, $\mathbf{k}$ and $\mathbf{p}'$ as input to make a DFS starting from the $\mathbf{p}'$-node (line 12). Note that this recursive call to PTreeDFS is performed on the subgraph $\mathcal{M}[\{Q_1, Q_2, \ldots, Q_l\}]$ of the input multi-layer graph $\mathcal{M}$ induced by $Q_1, Q_2, \ldots, Q_l$ (defined in Section 3.1), i.e., $\mathcal{M}_{\mathbf{k}}[Q_l]$, instead of on $\mathcal{M}$ because the $(\mathbf{k}, \mathbf{p}')$-core can be correctly computed on $\mathcal{M}_{\mathbf{k}}[Q_l]$ according to Implication I2, thereby reducing computational overhead. Finally, PTreeDFS returns $\{Q_1, Q_2, \ldots, Q_l\}$ (line 13).

KPTreeDFS is a recursive procedure to realize a DFS on the KP-tree, which is very similar to PTreeDFS. When the $\mathbf{k}$-node in the KP-tree is traversed, we start a DFS on the P-tree nested in the $\mathbf{k}$-node by calling PTreeDFS$(\mathcal{M}, \mathbf{k}, \mathbf{0}^{l-1}, R)$ (line 15), where $\mathbf{0}^{l-1}$ represents the $(l-1)$-dimensional vector of zeros. The output $\{Q_1, Q_2, \ldots, Q_l\}$ of PTreeDFS$(\mathcal{M}, \mathbf{k}, \mathbf{0}^{l-1}, R)$ corresponds to the $(\mathbf{k}, \mathbf{0}^{l-1})$-core. If $Q_l = \emptyset$, all descendants of the $\mathbf{k}$-node need not be traversed according to Implication I1, so lines 17–21 can be skipped. If $Q_l \neq \emptyset$, any $(\mathbf{k}', \mathbf{p}')$-core in the descendants of the $\mathbf{k}$-node is completely contained in $\mathcal{M}[\{Q_1, Q_2, \ldots, Q_l\}]$ according to Implication I2. Thus, line 21 recursively calls KPTreeDFS with $\mathcal{M}[\{Q_1, Q_2, \ldots, Q_l\}]$ given as input.

The GCD+ algorithm simply calls KPTreeDFS$(\mathcal{M}, \mathbf{0}^l, R)$ to start the DFS traversal starting from the root of the KP-tree. All nonempty gCores discovered during the DFS are kept in the result set R.

D.2 Discussion on Alternatives of KP-tree

Except for KP-trees, there are alternative ways to represent the search space of the GCD problem. Similar to Galimberti et al. [12], the search space can be represented by a lattice, where a $(\mathbf{k}, \mathbf{p})$-core can have multiple parents. There are two main disadvantages to representing the search space by a lattice. First, a $(\mathbf{k}, \mathbf{p})$-core can be generated multiple times based on its parents [12]. Therefore, a complicated strategy needs to be designed to avoid duplicated generation. Second, people usually store some useful information about the traversed part of the search space to speed up the subsequent search process. Since the structure of the lattice is much more complex than the KP-tree, more information needs to be stored, thereby the storage costs are higher. Both of these disadvantages are avoided by KP-trees.

E SUPPLEMENTARY EXPLANATION TO GCS+ (ALGORITHM 3)

In this section, we give some supplementary explanations to the KP-tree-based GCS algorithm GCS+ (Algorithm 3) introduced in Section 6.2.

As stated in Section 6.2, to retrieve the $(\mathbf{k}, \mathbf{p})$-core from the KP-tree, GCS+ first search the $\hat{\mathbf{p}}$-node in the P-tree nested in the $\mathbf{k}$-node of the KP-tree, where $\hat{\mathbf{p}}[i]$ is the smallest element in F_i (Equation 1) that is not less than $\mathbf{p}[i]$ for $i = 1, 2, \cdots, l-1$, and then compute the core represented by the $\hat{\mathbf{p}}$-node as the $(\mathbf{k}, \mathbf{p})$-core.

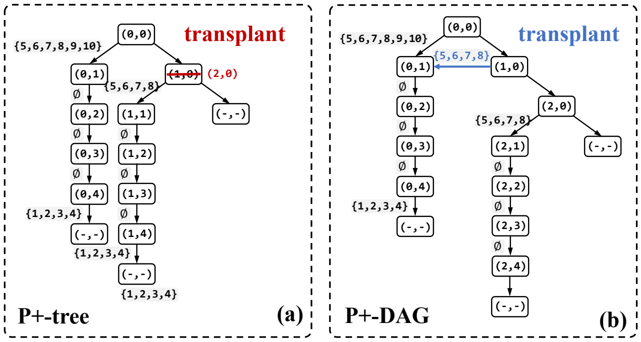


Figure 15: Illustration of subtree transplant in (a) subtree merge and (b) subtree merge.

The $\hat{\mathbf{p}}$-node search process is described in Procedure Search in Algorithm 3. It repeatedly calls Procedure Forward to find the path from the root of the P-tree, i.e., the $\mathbf{0}^{l-1}$-node, to the $\hat{\mathbf{p}}$-node. Let N be the node under investigation, and let i be a counter. Initially, N is set to the root of the P-tree, and i is set to 1. Procedure Forward works as follows: Let $\mathbf{p}'$ be the vector (in the fractional form) associated with N. If $\mathbf{p}'[i] < \mathbf{p}[i]$, Procedure Forward finds the child N' of N such that the vectors associated with N and N' are different in their i-th elements (line 14), and then returns N' and i (line 15). Otherwise, it returns N and $i+1$ (line 17). Procedure Search repeatedly calls Forward to update N and i until $i = l$ (lines 8–9). When $i = l$, N is eventually the $\hat{\mathbf{p}}$-node. The correctness of repeatedly applying Forward to find the path from the root of the P-tree to the $\hat{\mathbf{p}}$-node is guaranteed by the following invariant.

Lemma E.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 $\hat{\mathbf{p}}$-node in the P-tree. Let $\mathbf{p}'$ be the vector (in fractional form) associated with N'. If $i' = i+1$, we have that $\mathbf{p}'[j]$ is the smallest element in F'_j that is not less than $\mathbf{p}[j]$ for $j = 1, 2, \ldots, i$.*

After that, Procedure Recover (lines 18–25) is called in line 4 to collect the vertex sets on the leftmost path of the $\hat{\mathbf{p}}$-node down to a dummy leaf node and the union of these vertex sets is returned as the result. Theorem 6.2 ensures the correctness of Procedure Recover.

F SUPPLEMENTS FOR SUBTREE TRANSPLANT.

As we have discussed in Section 7.4, the DFS-order generation of P-tree nodes equipped with Theorem 7.10 enables a "*subtree transplant*" approach to reuse redundant subtree structures. In this section, we will show how subtree transplant works in detail.

Let N and N' be any two nodes in a P-tree, where N' is the rightmost child of N. The preceding subtrees rooted at N and N' are denoted as T and T', respectively. Once we have identified that $T \cong T'$, T and T' have the same structure but different vectors associated with their nodes. As shown in Equation (2), there is a bijection f from the nodes in T to the nodes in T'. Thus, in principle, we can update the vector associated with each node N in T to the vector associated with $f(N)$, and then transplant T to the position where T' is planted. That is why we name the approach "*subtree transplant*". Interestingly, we found that only the vector on the root of T has to be substituted with the vector on the root of T', and the correctness of applying Algorithm 3 to solve the GCS problem on the obtained P+-tree won't be affected (see Appendix H.4 of the full paper [24]). We then have T' obtained in $O(1)$ time.

Example. An illustration of subtree transplant is given in Figure 15 (a). Let N and N' be the nodes with vectors $(1,0)$ and $(2,0)$, and let T and T' be the preceding subtrees rooted at N and N', respectively. We have $T \cong T'$. To realize subtree elimination, we transplant T to the location of T' by substituting the vector on N with the vector $(2,0)$ on N'. When the $(\mathbf{k},\mathbf{p})$-core for $\mathbf{p} = (2,2)$ is queried, Algorithm 3 locates the node with vector $(1,2)$ and returns $\{1,2,3,4\}$.

Batch subtree transplant. The subtree transplant technique introduced above transplants one redundant subtree at a time. However, there may sometimes exist more than one redundant subtree. In this case, multiple redundant subtrees can be removed in one batch and unnecessary computations can be further reduced. In the following, we propose a "batch subtree transplant" approach to realize this.

Let $R_1, R_2, \ldots, R_n$ be the rightmost path starting from the node R_1 in a P-tree. For $i = 1, 2, \ldots, n$, let T_i be the preceding subtree rooted at R_i, and $\mathbf{p}_i$ be the vector associated with R_i. Of course, $\mathbf{p}_i$ is an immediate suffix successor of $\mathbf{p}_{i-1}$ for $i \geq 2$. Let k be the dimension at which $\mathbf{p}_1$ differs from $\mathbf{p}_2$. By Definition 5.2, we have $\mathbf{p}_2[k] = \mathbf{p}_1[k] + 1$, as well as $\mathbf{p}_{i+1}[k] = \mathbf{p}_i[k] + 1$ for $2 \leq i < n$. Let $\hat{\mathbf{p}}$ be the signature of T_1 (Definition 7.9). There must exist $m \in \{1, 2, \ldots, n\}$ such that $\mathbf{p}_m[k] = \hat{\mathbf{p}}[k]$. According to Theorem 7.10, we have $T_1 \cong T_2 \cong \cdots \cong T_m$, so $T_1, T_2, \ldots, T_{m-1}$ are redundant and therefore can be eliminated. By extending the subtree transplant method proposed in Section 7.4, the elimination of $T_1, T_2, \ldots, T_{m-1}$ as well as the construction of T_m can be simply realized in one subtree transplant process in $O(1)$ time, i.e., changing $\mathbf{p}_1$ to $\mathbf{p}_m$ and directly transplanting T_1 to the position where T_m is planted. In this way, we can avoid cascaded execution of subtree transplant, that is, T_1 is transplanted to the position of T_2; T_2 is transplanted to the position of T_3; … Finally, T_{m-1} is transplanted to the position of T_m.

G SUPPLEMENTS FOR SUBTREE MERGE.

G.1 Redundant subtree detection

In Section 7.5, we have proposed a subtree merge technique targeting at reducing the redundancy between preceding subtrees rooted at a node and any of its children. Specifically, we conceptually divide a P-tree into branches (Definition 7.11) and identify redundant subtrees within corresponding branches. After that, redundant subtrees are merged into one another.

A supplementary illustration of the branch is given in Figure 16. By extending Definition 7.11 into P+-tree, we can certainly use a similar subtree merge procedure to compact a P+-tree as to compact a P-tree. However, it is worth noticing that whether a subtree T_1 rooted at a node N_1 can be merged into the subtree T_2 rooted at a node N_2 in the P+-tree depends on the isomorphism between subtrees rooted at N_1 and N_2 in the original P-tree, say T'_1 and T'_2. In fact, $T_1 \cong T_2$ not necessarily guarantees $T'_1 \cong T'_2$. For ease of distinguishing, we say T_1 and T_2 are *strong isomorphic* if T'_1 and T'_2 are isomorphic.

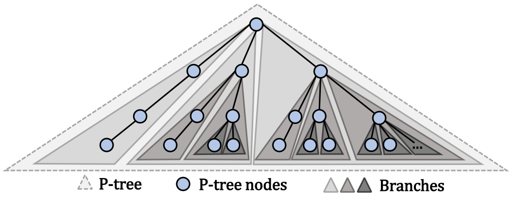


Figure 16: Illustration of branches.

Let us briefly describe the process of applying subtree merge to a P+-tree. Let N and N' be two nodes in a P+-tree, and N' be a child of N. We consider each pair of branches B_i^N and $B_i^{N'}$, where i ranges from $l - end0(\mathbf{p}')$ to $l-1$, and $\mathbf{p}'$ is the vector associated with N'. Let $R_0, R_1, R_2, \dots$ be the rightmost path in B_i^N where $R_0 = N$, and $R'_0, R'_1, R'_2, \dots$ be the rightmost path in $B_i^{N'}$ where $R'_0 = N'$. We use T_j and T'_k to denote the subtree rooted at R_j and R'_k, respectively. For all j and k, we test whether the subtrees rooted at R_j and R'_k are strong isomorphic. And if it is true, we merge T_j into T'_k by removing T and making R'_k a new child of R_{j-1} instead of R_j. Based on the signature (Definition 7.9) of T_j, a fast test method is given in the theorem below.

Theorem G.1. *Let $\mathbf{p}_j$ and $\mathbf{p}'_k$ be the vectors associated with R_j and R'_k, respectively. Let $\hat{\mathbf{p}}_j$ be the signature of T_j. T_j is strong isomorphic to T'_k if and only if*
(1) $\mathbf{p}_j[i] = \mathbf{p}'_k[i]$,
(2) $\mathbf{p}'_k[i'] \leq \hat{\mathbf{p}}_j[i']$ *for* $i' = 1, 2, \cdots, i-1$.

What is the order for testing whether the subtrees rooted at R_j and R'_k can be merged, i.e., T_j and T'_k are strong isomorphic? We have a very important observation: when T_j is merged into T'_k, all subtrees rooted at $R_{j'}$ for $j' > j$ are consequently merged into T'_k as they are contained in T_j. Therefore, we examine the subtree rooted at R_j in increasing order of j, and test if T_j and T'_k are strong isomorphic for each T'_k. Whenever subtree merge is applicable, we get the most succinct branch.

G.2 Pivot-based Optimization of Redundant Subtree Detection

Using Theorem G.1 to check the strong isomorphism between subtrees rooted at each pair of R_j and R'_k in branches B_i^N and $B_i^{N'}$ is still not efficient because of the following twofold limitations. (1) Assume the lengths of the rightmost paths in B_i^N and $B_i^{N'}$ are m and n, respectively. In the worst case (when no nodes on the rightmost path of B_i^N have its subtree strong isomorphic to the subtree rooted at nodes on the rightmost path of in $B_i^{N'}$), it takes $O(mn)$ time to run the testing procedure presented in Section G.1. In fact, many isomorphic subtree detection operations are unnecessary and therefore can be skipped. (2) To check the Condition (1) in Theorem G.1, we must know the vectors associated with both R_j and R'_k in advance, which requires the preceding subtrees rooted at R_j and R'_k are entirely generated. However, once we found the subtrees rooted at R_j and R'_k are strong isomorphic, the subtree of one of them in the P+-tree will be removed, and thereby the computation is wasted.

To overcome the above limitations, we devise a sufficient condition to determine the strong isomorphism between subtrees rooted at R_j and R'_k, which is shown in the following Theorem G.2. Notice that the theorem is applicable even before R'_k and the subtree rooted at R'_k has been generated.

Theorem G.2. *The subtree rooted at R_j is strong isomorphic to the subtree rooted at R'_k if*
(1) $\mathbf{p}_{j-1}[i] \leq \mathbf{p}'_{k-1}[i] < \mathbf{p}_j[i]$,
(2) $\mathbf{p}'_{k-1}[i'] \leq \hat{\mathbf{p}}_j[i']$ *for* $i' = 1, 2, \cdots, i-1$.

Based on the theorem, we devise a pivot-based optimization of the strong isomorphic subtree detection approach. Recall Property 4 in Section 7.2: for a vector $\mathbf{p}$ and any of its suffix successors $\mathbf{p}'$, we have $\mathbf{p} < \mathbf{p}'$. Therefore, for the vectors $\mathbf{p}_0, \mathbf{p}_1, \dots, \mathbf{p}_m$ associated with the nodes on the rightmost path in B_i^N, we have $\mathbf{p}_0 < \mathbf{p}_1 < \cdots < \mathbf{p}_m$. For the vectors $\mathbf{p}'_0, \mathbf{p}'_1, \dots, \mathbf{p}'_n$ associated with the nodes on the rightmost path in $B_i^{N'}$, we have $\mathbf{p}'_0 < \mathbf{p}'_1 < \cdots < \mathbf{p}'_n$. Then, for each node R'_k on the rightmost path of $B_i^{N'}$, we can define a node R_j on the rightmost path of B_i^N as the pivot node if j is the smallest integer such that $\mathbf{p}_j[i] > \mathbf{p}'_{k-1}[i]$. It is easy to see that the subtree T_j rooted at R_j is the only subtree rooted at a node on the rightmost path of B_i^N that satisfies Condition (1) in Theorem G.2. If T_j also satisfies Condition (2) in Theorem G.2, T_j is strong isomorphic to the subtree T'_k rooted at R'_k; Otherwise, no subtree rooted at a node on the rightmost path of B_i^N is strong isomorphic to T'_k.

A useful property of pivot nodes is given as follows: Let R'_{k_1} and R'_{k_2} be two nodes on the rightmost path of $B_i^{N'}$. Suppose the pivot nodes for R'_{k_1} and R'_{k_2} are R_{j_1} and R_{j_2}, respectively. If $k_1 < k_2$, we certainly have $j_1 \leq j_2$. Therefore, for all nodes on the rightmost path of $B_i^{N'}$, their pivot nodes can be found by scanning the nodes on the rightmost path of B_i^N only once. Subsequently, the time complexity of strong isomorphic subtree detection is reduced to $O(m+n)$ in the worst case.

G.3 Implementation.

A straightforward implementation of subtree merge is to first generate a P+-tree and then repeatedly apply subtree merge to the P+-tree to obtain a P+-DAG. However, such implementation usually performs lots of wasted computations as many generated subtrees are discarded later if they are merged into other subtrees. Therefore, like pushing subtree elimination into the P-tree generation process using subtree transplant (Section 7.4), we propose to push subtree merge into the P+-tree generation process to avoid the unnecessary generation of merged subtrees.

Remember that the nodes of a P+-tree are generated in a depth-first order. Such order leads to the following facts. (1) For each node N with vector $\mathbf{p}$, the branch B_i^N must be generated prior to its children N' with vector $\mathbf{p}'$ if $end0(\mathbf{p}') > i$ and therefore prior to the generation of $B_i^{N'}$. (2) For the nodes $R'_0, R'_1, \cdots$ on the rightmost path of $B_i^{N'}$, the preceding subtree rooted at R'_{k-1} must be generated prior to R'_k for $k > 1$. When R'_{k-1} and its preceding subtree have been generated, we are able to test whether the subtree T' rooted at R'_k is strong isomorphic to some subtree T in B_i^N using the pivot-based approach proposed in Section G.2. If $T \cong T'$, we merge T into T' by removing T and making R'_{k-1} to be the parent of T.

However, T' has not been generated yet at this time. To avoid generating T' from scratch, we make use of the isomorphism between T and T' and apply subtree transplant (see Section 7.4) to obtain T'. Specifically, we transplant T to the position where T' is planted. Although the vectors associated with the nodes in T and T' are different, we found that the correctness of applying Algorithm 4 to solve the GCS problem on the P+-DAG obtained after subtree transplant will not be affected (see Appendix M for detailed analysis).

Example. Figure 15 (b) illustrates the process of using subtree transplant to realize subtree merge. Let N_1, N_2, N_1', N_2' be the nodes with $\mathbf{p} = (0,0), (0,1), (1,0), (1,1)$, respectively. As we have already known that the subtree T rooted at N_2 is strong isomorphic to the subtree T' rooted at N_2', we transplant T to the position where T' is planted by simply making N_2 to be a child of N_1'. When the $(\mathbf{k}, \mathbf{p})$-core for $\mathbf{p} = (1,2)$ is queried, Procedure Search in Algorithm 3 returns the node associated with vector $(0,2)$. Finally, Algorithm 3 returns $\{1,2,3,4\}$ as the result.

Using the pivot-based approach to detect strong isomorphic subtrees (Section G.2) and the above subtree-transplant-based subtree merge approach bring extra $O(\sum_{i=2}^{l-1} \prod_{j=1}^{i} |F_j|)$ time overhead to the P+-tree generation process (see Appendix N for the complexity analysis). However, it prevents a lot of redundant subtrees from being generated and obtains a P+-DAG directly, which always takes much less time in practice.

H DATASETS CONSTRUCTION

We select 3 real-world pillar multi-layer graphs and construct 7 general multi-layer graphs using publicly available datasets as presented in Table 1. The details are given as follows:

SacchCere [12], **ObamaInIsrael** [28] and **Friendfeed** [9] are multiplex networks. **SacchCere** has 7-layers describing different types of genetic interactions between genes in Saccharomyces cerevisiae [12]. **ObamaInIsrael**[3] has 3 layers representing re-tweeting, mentioning, and replying relationships among Twitter users during the visit to Israel by Barack Obama in 2013 [28]. **Friendfeed**[4] has 3 layers defined by commenting, liking and following interactions among users of Friendfeed collected over two months[9].

DBLP is built from the DBLP (four-area) dataset [33], consisting of an author co-authoring layer and a term similarity layer. We use frequently appeared venues for each term as a feature to compute the Tanimoto similarities between pair-wise terms. Two terms are connected in the term similarity layer if their similarity is no less than 0.5. An author is connected to a term through a cross-layer edge if he (or she) has published a paper containing the term. By default, the author layer is of users' interest.

6-NG and **9-NG** are extracted from the 20-Newsgroup dataset[5] basically following [27]. Both graphs have 5 news similarity layers. Specifically, 6-NG consists of news from 6 newsgroups. From each newsgroup, $\{100, 125, 150, 175, 200\}$ distinct news are randomly selected and fed into 5 layers, respectively. After computing the cosine similarity between *tf-idf* vectors of pair-wise news, we regard two news are closely related if their similarity is at least 0.5 or one falls into the other's top-5 similar news. Each news is connected to its closely related news through either intra-layer edges or cross-layer edges. 9-NG is built in the same way but using news from 9 newsgroups.

Twitter is a 3-layer graph describing the co-occurring relationships between components of tweets. Following the idea in [32] with slight modifications, we collect about 20000 tweets concerning 4 hot events happened in 2016[6]: Rio2016, Election2016, PokemonGo and Brexit from Twitter[7]. Then, three types of key components, namely *hashtags*, *keywords* and *mentions*, are extracted from each tweet and fed into 3 layers, respectively. Every pair of components are connected by an edge (either an intra-layer edge or a cross-layer edge) if they have co-occurred in a tweet. By default, the mentions layer is of users' interest.

Movie is a 4-layer graph describing relationships among movies and casts. Specifically, we collect information about 68000 movies from TMDB[8], including the overview, similar movies list, recommended movies list and casts. Three movie similarity layers and a cast cooperation layer are thereby constructed. We compute the cosine similarity of the overview of pair-wise movies and connect two movies in the first movie similarity layer if one falls into the other's top-5 similar movies or their similarity is at least 0.5. In the second and third movie similarity layers, each movie is connected to the ones in its top-10 similar movies list and top-10 recommended movies list, respectively. Two casts are connected in the cast cooperation layer if they have cooperated in at least 2 movies. Each cast is connected to his/hers top-5 most popular movies through cross-layer edges. In the experiment, we randomly selected the movie similarity layer constructed based on the similar movies list as the layer of users' interest.

Aminer-5 and **Aminer-10** [35] are both 5-layer graphs extracted from the DBLP-Citation-network V13 dataset from Aminer[9]. We select papers published in 2011–2015 and 2011–2020 to construct the two graphs, respectively. Each graph contains three paper layers, an author layer and a venue layer. After computing the cosine similarity of the abstract of pair-wise papers, we build a paper similarity layer where two papers are connected if one falls into the other's top-5 similar papers or their similarity is at least 0.5. Besides, a paper citation layer and a paper co-citation layer are also built. Note that two papers are connected in the co-citation layer only if they have co-cited at least 3 papers. The author layer is defined by the co-authoring relationships between authors. We measure the closeness between venues by the number of their paper citations. Two venues are connected in the venue layer if one has cited the other at least 5 (resp. 10) times in Aminer-5 (resp. Aminer-10). With cross-layer edges, each paper is connected to its authors and the venue it has been published in. During the experimental evaluation, we randomly selected the paper similarity layer constructed based on the similarity of the abstract as the layer of users' interest.

[3] https://manliodedomenico.com/data.php

[4] http://multilayer.it.uu.se/datasets.html

[5] http://qwone.com/~jason/20Newsgroups/

Table 2: Intra-layer comparison on DBLP

Model	Size	Degeneracy	Den.	Clu. Coe.
$(\mathbf{k},\mathbf{p})$-core	454	**26**	7.2	**0.858**
$(10,\mathcal{P})$-core	10514	15	2.777	0.238
$(15,\mathcal{P})$-core	6161	12	3.074	0.215
$(20,\mathcal{P})$-core	4111	9	3.316	0.209
$(25,\mathcal{P})$-core	2950	9	3.489	0.205
$(30,\mathcal{P})$-core	2269	9	3.539	0.202
$(40,\mathcal{P})$-core	1427	9	3.702	0.206
$(50,\mathcal{P})$-core	989	9	3.78	0.218
$(100,\mathcal{P})$-core	202	7	3.505	0.276

Table 3: Intra-layer closeness comparison on Twitter

Model	Size	Degeneracy	Den.	Clu. Coe.
$(\mathbf{k},\mathbf{p})$-core	133	**8**	3.368	0.609
$(3,\mathcal{P}_1)$-core	1376	**8**	1.413	0.229
$(5,\mathcal{P}_1)$-core	343	**8**	2.14	0.272
$(10,\mathcal{P}_1)$-core	39	7	3.462	0.457
$(15,\mathcal{P}_1)$-core	16	7	**4.375**	0.723
$(10,\mathcal{P}_2)$-core	1398	**8**	1.386	0.213
$(15,\mathcal{P}_2)$-core	562	7	2.004	0.232
$(20,\mathcal{P}_2)$-core	345	7	2.336	0.274
$(25,\mathcal{P}_2)$-core	217	7	2.7	0.313
$(30,\mathcal{P}_2)$-core	165	7	2.939	0.347
$(40,\mathcal{P}_2)$-core	112	6	2.964	0.393
$(50,\mathcal{P}_2)$-core	63	6	2.603	0.38
$(40,\{\mathcal{P}_1,\mathcal{P}_2\})$-NMC	3517	**8**	1.021	0.209
$(50,\{\mathcal{P}_1,\mathcal{P}_2\})$-NMC	3259	**8**	1.031	0.207
$(100,\{\mathcal{P}_1,\mathcal{P}_2\})$-NMC	2075	**8**	1.098	0.231
$(200,\{\mathcal{P}_1,\mathcal{P}_2\})$-NMC	1156	**8**	1.196	0.343
$(400,\{\mathcal{P}_1,\mathcal{P}_2\})$-NMC	470	5	0.691	**0.785**

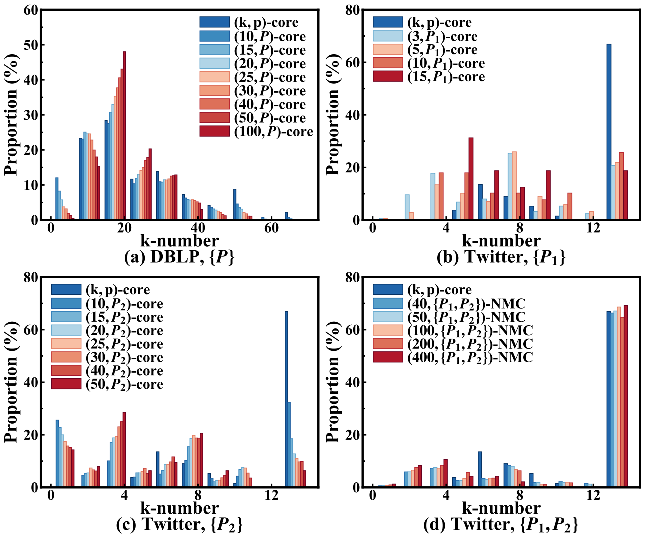


Figure 17: Distribution of the k-numbers of vertices in the the $(\mathbf{k},\mathbf{p})$-core and meta-path-based cores on DBLP and Twitter.

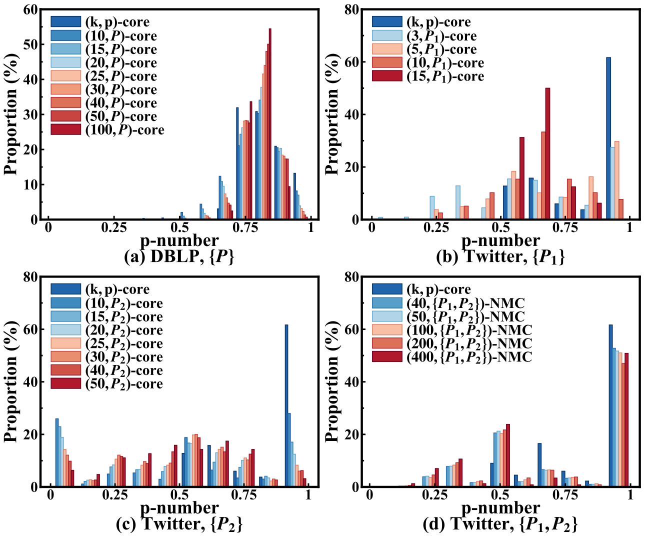


Figure 18: Distribution of the p-numbers of vertices in the the $(\mathbf{k},\mathbf{p})$-core and meta-path-based cores on DBLP and Twitter

I EFFECTIVENESS COMPARISON WITH META-PATH-BASED MODELS.

In this experiment, we conducted a comprehensive comparison between the gCore model and meta-path-based models, including $(k,\mathcal{P})$-core [11] and (k,Ψ)-NMC [18], in terms of their effectiveness in characterizing closely interconnected vertices. To achieve this, we compare the closeness of vertices in the $(\mathbf{k},\mathbf{p})$-core (gCore) and different versions of $(k,\mathcal{P})$-core as well as (k,Ψ)-NMC by varying k and used meta-paths on datasets DBLP and Twitter. Closeness under both the direct interactions represented by the target layer and the indirect interactions represented by other layers are considered. It should be noted that the existing algorithms proposed in [11] and [18] aim at identifying $(k,\mathcal{P})$-cores and (k,Ψ)-NMCs that contain a given query vertex, respectively. Therefore, to accomplish the comparison, select the vertex with the highest degree within the gCore and subsequently execute the $(k,\mathcal{P})$-core/(k,Ψ)-NMC search algorithm [11, 18]. Notice that the implementations of the $(k,\mathcal{P})$-core search algorithms are obtained from [37].

The same as the parameter settings used in Section 8.2.2, we set $\mathbf{k}=(10,10)$ and $\mathbf{p}=(0.7)$ for DBLP, and set $\mathbf{k}=(5,5,5)$ and $\mathbf{p}=(0.5,0.5)$ for Twitter. To adopt the $(k,\mathcal{P})$-core and (k,Ψ)-NMC models on the general multi-layer graph, we suppose vertices in each layer have a distinct type, and the type of vertices on the i-th layer is denoted by t_2. The DBLP dataset consists of two layers. Therefore, we use the meta-path $\mathcal{P}=\mathrm{t}_2\rightarrow\mathrm{t}_1\rightarrow\mathrm{t}_2$ and $\Psi=\{\mathcal{P}\}$ on DBLP. Twitter has three layers, and we set $\mathcal{P}_1=\mathrm{t}_3\rightarrow\mathrm{t}_1\rightarrow\mathrm{t}_3$, $\mathcal{P}_2=\mathrm{t}_3\rightarrow\mathrm{t}_2\rightarrow\mathrm{t}_3$, and $\Psi=\{\mathcal{P}_1,\mathcal{P}_2\}$. Notice that all these meta-paths are commonly used in ML-CSM works on HINs. Furthermore, to perform a more in-depth comparison, we varying k in $\{3,5,10,15,20,25,30,40,50,100,200,400,800\}$.

Unfortunately, we found that the basic $(k,\mathcal{P})$-core model often result in significantly large result, which covers more than half of

[6] https://blog.twitter.com/en_us/a/2016/thishappened-in-2016
[7] https://www.twitter.com/
[8] https://www.themoviedb.org/
[9] https://www.aminer.cn/citation

the entire vertices when $k = 800$ for DBLP and $k = 100$ for Twitter. These resultant vertices obviously exhibit sparse interactions. To ensure a meaningful comparison, we employ the vertex-disjoint $(k, \mathcal{P})$-core, which has been proven to exhibit the highest cohesiveness [11], for our analysis. Besides, all results with a size large than 50% of the total number of target vertices are also discarded.

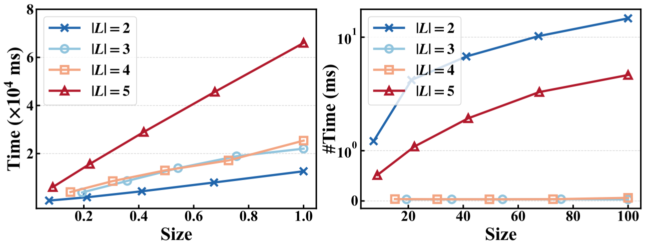
Closeness under direct interactions. We measure the closeness of vertices in terms of their direct interactions by the classical metrics including degeneracy, density (abbreviated as "den."), and clustering coefficient (abbreviated as "clu. coe."). The result obtained on DBLP and Twitter are presented in Table 2 and Table 3, respectively. It is evident that, in most cases, the $(\mathbf{k}, \mathbf{p})$-cores achieve higher degeneracy, density, and clustering coefficient compared to other meta-path-based cores. This clearly demonstrates the high cohesiveness of the $(\mathbf{k}, \mathbf{p})$-core under direct vertex interactions. However, as we have pointed out in Section 1, meta-path-based models completely overlook these direct relationships between homogeneous entities, while only focusing on the ones between heterogeneous vertices (vertices in different layers in this case), the results obtained using these models exhibit much sparser connections.

Closeness under indirect interactions. We measure the closeness of vertices exhibited on other layers with metrics **k-number** and **p-number** introduced in Section 8.2.2. As a representative, only the k-number and p-number w.r.t. G_0 are present.

The distributions of the k-numbers are given in Figure 17. We can observe that all meta-based models contain a proportion of vertices with small k-numbers. In the worst case, the vertex-disjoint $(k, \mathcal{P}_2)$-core and (k, Ψ)-NMC on Twitter covers $40\% - 50\%$ and $13\% - 20\%$ vertices with k-number smaller than 5 for all k. These vertices show weak interactions with other vertices w.r.t. G_0. This is because meta-path instances cannot capture the cohesiveness of their intermediate vertices, which has been shown to have a great enhancement to the cohesion among target vertices. Fortunately, these vertices are well excluded by the $(\mathbf{k}, \mathbf{p})$-core.

We also have some investigations concerning meta-path-based models, which will be given below:

(1) In most cases, the number of vertices with a small k-number in the $(k, \mathcal{P})$-core becomes smaller as k increases (see Figure 17 (a), (b), and (c)). The reason is as follows. A larger k requires a larger number of meta-path instances to contribute to the degree of each target vertex, indicating more cohesive interactions between the target vertices and the intermediate vertices involved in these meta-path instances. In the context of GMGs, this also implies denser inter-layer connections between the target vertices and their cross-layer neighbors. In real-world graphs, these dense inter-layer connections can reflect the closeness of the involved cross-layer neighbors to some extent. For example, in the DBLP dataset, which consists of an author-collaboration layer and a term similarity layer, the cross-layer edges represent the connections where an author has published a paper related to a specific term. When a group of authors exhibits a cohesive connection to a group of terms, these terms are likely to be similar and form a cohesive subgraph in the term-similarity layer. Consequently, each author tends to have a larger k-number. Nevertheless, the intra-layer cohesiveness of the terms cannot be fully captured by the inter-layer edges, which explains why there are still numerous vertices with small k-numbers in the $(k, \mathcal{P})$-core.



(a) GCS (b) GCS+-EM

Figure 19: Scalability of (a) GCS and (b) GCS+ with varying the number of layers and the number of vertices.

(2) By comparing Figure 17 (b) and (c), we can observe that vertices in the $(k, \mathcal{P}_1)$-core tend to have larger k-numbers than those in the $(k, \mathcal{P}_2)$-core. This is because $\mathcal{P}_1$ considers the connections between target vertices through vertices in G_0. The meta-path instances of $\mathcal{P}_1$ contributing to the $(k, \mathcal{P}_1)$-core reflect the closeness of vertices in G_0 at a certain level. Therefore, the target vertices in the $(k, \mathcal{P}_1)$-core have a large k-number in terms of G_0. On the other hand, instances of $\mathcal{P}_2$ do not directly capture this closeness.

(3) From Figure 17 (d), we can observe that vertices in the (k, Ψ)-NMC exhibit a distribution most similar to that of the $(\mathbf{k}, \mathbf{p})$-core. The (k, Ψ)-NMC significantly outperforms the $(k, \mathcal{P})$-core (see Figure 17 (b) and (c)) in terms of the k-number due to the following two-fold reasons. Firstly, the (k, Ψ)-NMC considers two meta-paths simultaneously. Secondly, it employs the basic $(k, \mathcal{P})$-core model rather than the vertex-distinct version. The basic $(k, \mathcal{P})$-core model seems to achieve a better result here because it imposes a less stringent constraint compared to the vertex-disjoint core on the cross-layer cohesiveness. As a result, it may include more vertices that exhibit intra-layer cohesiveness. To further support this, we compute the density of the bipartite graph comprised by the $(k, \mathcal{P}_1)$-core (resp. the (k, Ψ)-NMC) and its cross-layer neighbors in G_0, which are 3.39 and 4.68, respectively. This further verifies that the vertex-disjoint core places more emphasis on cross-layer connections, which may sometimes contradict the intra-layer cohesiveness.

However, it should be noted that although (k, Ψ)-NMC performs well in terms of the k-number measure, the resultant subgraph is pretty large and have a very low density in the target layer, as shown in Table 3. Besides, we have observed in the experiment that selecting an appropriate value for k to obtain a meaningful result in this model is quite challenging.

Figure 18 shows the results on p-numbers. Since the p-numbers exhibit a similar distribution to the corresponding k-numbers under the 4 experimental settings, we will not provide further elaboration on the analysis.

J SCALABILITY TEST

In this section, we evaluate the scalability of the algorithms proposed in this paper, including GCS, GCS+, and TEM, which has exhibited the best performance in solving the GCI problem.

Scalability of GCS and GCS+. To evaluate the scalability of GCS and GCS+, we test the runtime of GCS and GCS+ using different versions of Aminer-10 dataset obtained by selecting a variable

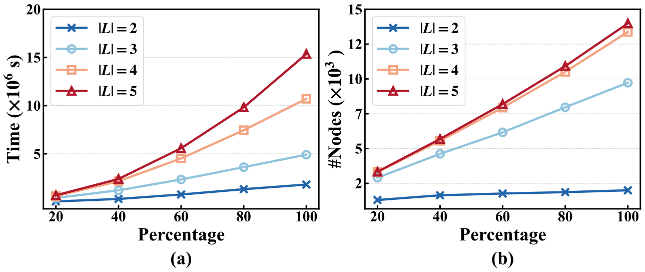


Figure 20: Scalability of TEM in terms of (a) runtime and (b) scale with varying the number of layers and the number of vertices.

number of layers and subsets of vertices. Layers are selected by keeping the layer of users' interest, i.e., G_l, and adding layers G_i in increasing order of i. Then, for each selected subset of layers, we randomly sample 20%, 40%, 60%, 80%, and 100% of the vertices from G_l. By combining their corresponding (cross-layer) induced subgraphs on each layer, we obtain five new versions of the Aminer-10 dataset. Notice that the size of the obtained multi-layer graph may not necessarily align with the selected percentage of vertices. We normalize the sizes of all versions of multi-layer graphs with the same set of layers to 1.

Figure 19 presents the runtime of GCS and GCS+ with respect to the size of the multi-layer graph and the number of layers $|L|$. Notice that the performance of GCS+-EM is tested as a representative of the GCS+ algorithm. We have the following observations. 1) GCS and GCS+ exhibit linear and sub-linear scalability with the size of the multi-layer graph, respectively, when the number of layers is fixed. 2) Introducing new layers may not necessarily increase the runtime of GCS, see the line corresponding to $|L| = 3$ and 4 in Figure 19 (a). This is because the constraint imposed on the new layer can facilitate the peeling process and result in faster convergence. The saved time compensates for the time spent on checking newly involved vertices and edges. 3) The impact of introducing new layers on the efficiency of GCS+ is uncertain. On one hand, the P-tree becomes larger, and more nodes need to be visited to search for the target **p**-node. On the other hand, the constraint imposed on the new layer leads to a smaller gCore and consequently a smaller core-retrieval time. Nevertheless, GCS+ always shows orders of magnitude speed-up compared with GCS.

Scalability of TEM. To evaluate the scalability of TEM, we test its runtime and the scale of the generated KP-tree using different versions of Aminer-10 dataset obtained in the same way as the one used in the scalability evaluation of GCS and GCS+.

Figure 20 reports the runtime and the scale of the generated KP-trees with respect to the percentage of selected vertices on the layer of users' interest and the number of layers $|L|$. From (a), we obtain the following observation: 1) When the number of layers is fixed, the runtime of TEM increases at a larger pace as the selected percentage increases. This is because both the size of the multi-layer graph and the number of **p**-nodes to be computed in the KP-tree (see (b)) grow. 2) Adding a layer leads to a varying increase in the runtime of TEM, which is determined by the specific characteristics of the newly added layer.

And from (b), we can observe that: 1) When fixing the number of layers, the number of **p**-nodes in the KP-tree scales linearly in the number of vertices on the layer of users' interest. 2) Adding a layer leads to a varying increase in the number of **p**-nodes in the KP-tree. This is because the increment is determined by both the degeneracy of the new layer and the type of cross-layer mappings between vertices in G_l and the new layer. These two factors vary across different layers, leading to varying increments.

K CASE STUDY ON LAST.FM

We apply the gCore model to discover users sharing friendships and similar musical tastes on Last.Fm, an online music-sharing platform (https://www.last.fm/). The dataset is obtained from [6], involving 1892 users and 17632 artists. We construct a 3 layer GMG using the dataset, which consists of an artist similarity layer, a user friendship layer, and a user similarity layer. We use user-labeled tags as features to compute the Tanimoto similarities between pair-wise artists. Two artists are considered adjacent in the artist similarity layer if their similarity is at least 0.5. The user friendship layer describes the friendships between users, which are directly obtained from the original dataset. Additionally, we incorporate the labels of users' frequently listened artists as features to calculate the pair-wise similarities between uses. Two users are adjacent in the user similarity layer if their similarity is equal to or exceeds a threshold of 0.5. Among these layers, the user similarity layer is of our interest.

We compute the gCore on the graph with $k = (5, 5, 5)$ and $p = (0.5, 1)$. The result is a group of 63 users. From the definition of gCore, we know there exists a 5-core Q on the artist layer that covers at least 50% liked artists for each user in the gCore. As the users in the dataset are anonymous, we only show the artists in Q in Table 4. Since Q is a 5-core, the artists within it display similarities in terms of user-labeled tags. Furthermore, in terms of practical significance, all of these artists are internationally renowned female musicians who excel in music genres like pop and dance. Consequently, they are well-suited for recommendations to users within the gCore who have not frequently listened to them before.

Table 4: Artists obtained from Last.Fm.

Artists
Rihanna, Britney Spears, Lady Gaga, Katy Perry, Christina Aguilera, Kylie Minogue, Madonna

L DISCUSSIONS ON KP-TREE MAINTENANCE

Maintaining the KP-tree against dynamic changes in a multi-layer graph is more challenging compared to maintaining indexes of cohesive subgraphs proposed for single-layer graphs from the following two aspects:

(1) Multi-layer graphs introduce additional dimensions of update. In addition to the traditional operations such as adding or removing nodes and edges, multi-layer graphs involve operations such as adding or removing layers, as well as intra-layer and cross-layer edges.
(2) Changes in one layer of a multi-layer graph can have cascading effects on other layers through cross-layer edges.

Due to the increased update operations and the inter-layer dependencies, efficient KP-tree maintenance requires a thorough investigation of its structure and careful consideration of all possible cases. Next, we outline the approach to address the problem, leaving the details in further work. Notice that the insertion or deletion of an isolated vertex can be equivalently represented as the insertion or deletion of all its incident edges, regardless intra-layer edges or cross-layer edges, we consider the update to the KP-tree against layer insertion/deletion and edge insertion/deletion. For clarity, we refer to the layer of users' interest, i.e., G_l as the primary layer, and the remaining layers as the secondary layers.

Layer update. Update to layers can be categorized into three cases: (a) the primary layer is changed; (b) an existing secondary layer is deleted; and (c) a new secondary layer is inserted. We will discuss the three cases separately.

Case (a). When the primary layer is changed, for example, in an academic multi-layer graph shown in Figure 2, users shift their attention to the connections between papers, rather than the previously focused relationships between authors, the KP-tree has to be recomputed. This is because the vertices stored in the index are changed from authors to papers, and the cross-layer edges considered are also changed from author-paper edges to paper-author edges.

Case (b). When an existing secondary layer is deleted, w.l.o.g., let us suppose G_i, where $1 \leq i < l$, is removed, it can be seen as all constraints imposed on G_i through $\mathbf{k}[i]$ and $\mathbf{p}[i]$ are removed. Thus, all $\mathbf{k}$-nodes with $\mathbf{k}[i] > 0$ and $\mathbf{p}$-nodes with $\mathbf{p}[i] > 0$ can be removed from the KP-tree. Specifically, in the KP-tree, for each $\mathbf{k}$-node with $\mathbf{k}[i] = 0$, we remove the entire subtree rooted at its child $\mathbf{k}'$-node, where $\mathbf{k}$ and $\mathbf{k}'$ differ at the i-th component. Then, for all remaining $\mathbf{k}$-node, we refine its vector by removing the i-th component. For example, the $(0, 1, 0)$-node becomes the $(1, 0)$-node if $i = 0$. Next, we perform a similar refinement to the P-tree inside each $\mathbf{k}$-node to reflect the removal of G_i.

Notices that if the removed layer is G_{l-1}, all leftmost paths of the original KP-tree and their associated vertex sets are removed in previous steps. Consequently, we have to recompute the vertex set associated with edges along each leftmost path to ensure the correctness of Theorem 6.2. Fortunately, it can be implemented efficiently as each core represented by the refined KP-tree can be obtained by searching the original KP-tree, rather than computing it from scratch.

Case (c). When a new secondary layer is inserted into an l-layer multi-layer graph, for the sake of simplicity, let us denote the newly added layer as G_l, and the primary layer as G_{l+1}. Contrary to Case (b), for each $\mathbf{k}$-node in the KP-tree, we refine its vector by inserting an 0 before the last component to reflect the addition of G_l. For example, the $(0, 1)$-node will become the $(0, 0, 1)$-node after a layer insertion. Accordingly, a new leftmost path is added to the refined node to represent all possible $\mathbf{k}'$-nodes with $\mathbf{k}' = (0, i, 1)$ for $i > 0$.

Next, we perform a similar refinement to each $\mathbf{p}$-node of the P-tree inside each $\mathbf{k}$-node to reflect the insertion of G_l. Notice that all vertex sets associated with edges on the leftmost paths of the refined KP-tree are missing, and therefore have to be computed. Nevertheless, the recomputation requires less time than constructing a new P-tree from scratch because the core represented by each $\mathbf{p}$-node in the previous P-tree, which is equal to the core represented by its corresponding refined node in the new P-tree, can be reused.

Edge update. Update to edges can be categorized into two cases: (a) insertion/deletion of an intra-layer edge; (b) insertion/deletion of a cross-layer edge. We will discuss the two cases separately. Particularly, we focus on edge insertion in each case as a representative. The solution to deal with edge deletion can be mainly derived from edge insertion cases and will be studied in future work.

Case (a). Without loss of generality, we suppose an edge (u, v) in $G_i = (V_i, E_i)$, where $1 \leq i \leq l$, is inserted. Saríyüce et al. have pointed out that the core number of any vertex in V_i changes at most 1 [30]. Let k be the minimum of the core number of u and v. It can be easily derived that all P-trees inside the $\mathbf{k}$-node, where $\mathbf{k}[i] > k + 1$, will not be affected. Thus, only the rest of the P-trees may need to be updated.

If G_i is a secondary layer, all cores represented by $\mathbf{p}$-nodes such that $\mathbf{p}[i] = 0$ are kept unchanged as they are computed without referencing G_i. We therefore update the rest parts of the P-tree. Specifically, for each $\mathbf{p}$-nodes with $\mathbf{p}[i] = 0$, we recompute the subtree rooted at its child $\mathbf{p}'$-node, where $\mathbf{p}$ and $\mathbf{p}'$ differ at the i-th component. Unfortunately, the situation becomes more tricky when G_i is the primary layer, and currently, we do not have an efficient way to identify the affected part of the P-tree in this situation. A possible solution would be to recompute the entire P-tree. However, inspired by the idea in the KP-Index maintenance studied in [40], we believe there is a way to reduce the number of recomputed cores. This will be the main focus of our future work.

Case (b). Without loss of generality, we suppose an cross-layer edge (u, v), where $u \in G_l$ and $v \in G_i$ $(1 \leq i \leq l)$, is inserted. Let k_1 be the core number of u in G_l, and k_2 be the core number of v in G_i. It's obvious that all P-tree insides the $\mathbf{k}$-node, where $\mathbf{k}[l] > k_1$ or $\mathbf{k}[i] > k_2$, will not be affected. Hence, only the rest of the P-trees may need to be updated.

For any $(\mathbf{k}, \mathbf{p})$-core Q containing u, the direct effect of inserting (u, v) is enlarge $G_i[Q]$. Therefore, we can apply a similar approach to the one used in dealing with the secondary layer edge insertion in Case (a) to handle this case. Specifically, for each $\mathbf{p}$-nodes with $\mathbf{p}[i] = 0$, we recompute the subtree rooted at its child $\mathbf{p}'$-node, where $\mathbf{p}$ and $\mathbf{p}'$ differ at the i-th component. Certainly, we will make efforts to find a way to further reduce the number of reduced cores under this case.

Besides the above discussions, we also highlight the following two directions that will be explored in our future work: (1) Finding effective strategies to group and handle multiple edge insertions and deletions in a single batch. (2) Assessing the circumstances under which the update to the KP-tree might fall behind the recomputation.

M MISSING PROOFS

This section provides the proofs for properties, lemmas and theorems proposed in this paper. For ease of notation, we let $d_G(v)$ denote the degree of vertex v in G.

M.1 Properties

Proof. **(Property 1)** We prove the property by contradiction. Suppose Q and Q' are two distinct $(\mathbf{k}, \mathbf{p})$-cores of $\mathcal{M}$. By Definition 3.3, we have (1) both Q and Q' are $\mathbf{k}[l]$-core; (2) there exist $\mathbf{k}[i]$-cores Q_i and Q_i' on $G_i[Q]$ such that $\phi(v, Q_i) \geq \mathbf{p}[i]$ for $v \in Q$ and $\phi(v', Q_i') \geq \mathbf{p}[i]$ for $v' \in Q'$, respectively; (3) both Q and Q' are maximal. Let $Q'' = Q \cup Q'$. We have $d_{G_l[Q'']}(v) \geq$

$\min\{d_{G_l[Q]}(v), d_{G_l[Q']}(v)\} \geq \mathbf{k}[l]$ for $v \in Q''$. Therefore, Q'' is a $\mathbf{k}[l]$-core. Similarly, we have $Q''_i = Q_i \cup Q'_i$ is a $\mathbf{k}[i]$-core on $G_i[Q'']$ for $1 \leq i < l$. Besides, it holds that for $v \in Q''$, $\phi(v, Q''_i) \geq \min\{\phi(v, Q_i), \phi(v, Q'_i)\} \geq \mathbf{p}[i]$. By Definition 3.3, Q'' is also a $(\mathbf{k}, \mathbf{p})$-core. It contradicts with the hypothesis of the maximality of Q and Q'. Thus, the property holds. □

Proof. **(Property 2)** Let Q' and Q be the $(\mathbf{k}_1, \mathbf{p})$-core and the $(\mathbf{k}_2, \mathbf{p})$-core, respectively. By Definition 3.3, we have (1) $d_{G_l[Q]}(v) \geq \mathbf{k}_2[l] \geq \mathbf{k}_1[l]$ for $v \in Q$; (2) there exists a $\mathbf{k}_2[i]$-core Q_i on $G_i[Q]$ such that $\phi(v, Q_i) \geq \mathbf{p}[i]$ for $v \in Q$. Certainly, Q_i is also a $\mathbf{k}_1[i]$-core. According to the definition of the $(\mathbf{k}_1, \mathbf{p})$-core and the maximality of Q', we have $Q \subseteq Q'$. Thus, the property holds. □

Proof. **(Property 3)** Let Q' and Q be the $(\mathbf{k}, \mathbf{p}_1)$-core and the $(\mathbf{k}, \mathbf{p}_2)$-core, respectively. By Definition 3.3, we have (1) $d_{G_l[Q]}(v) \geq \mathbf{k}[l]$ for $v \in Q$; (2) there exists a $\mathbf{k}[i]$-core Q_i on $G_i[Q]$ such that $\phi(v, Q_i) \geq \mathbf{p}_2[i] \geq \mathbf{p}_1[i]$ for $v \in Q$. According to the definition of the $(\mathbf{k}, \mathbf{p}_1)$-core and the maximality of Q', we have $Q \subseteq Q'$. Thus, the property holds. □

Proof. **(Property 4)** Let T be the P+-tree and T^c be the original P-tree of T. Let N'' be the child of N in T^c whose vector $\mathbf{p}''$ is an immediate suffix successor of $\mathbf{p}$ and differs from $\mathbf{p}$ only in the i-th element. Suppose $P = N_0, N_1, \cdots$ are nodes on the rightmost path of N'' in T^c, where $N_0 = N''$, and $\mathbf{p}_j$ is the vector associated with N_j for $j \in \mathbb{N}$. The structure of T^c ensures $\mathbf{p}_j$ is an immediate suffix successor of $\mathbf{p}_{j-1}$ for $j > 1$, and they differ in only the i-th element. As N' must be one of the nodes in P, we certainly have $\mathbf{p}'$ and $\mathbf{p}$ are different only the i-th element, $\mathbf{p}[i] < \mathbf{p}'[i]$ and $\mathbf{p}[i'] = \mathbf{p}'[i'] = 0$ for $i' > i$. Thus, the property holds. □

M.2 Lemmas

Proof. **(Lemma 5.1)** We construct each element $\hat{\mathbf{p}}[i]$ of $\hat{\mathbf{p}}$ as follows. Let $f_{i,1}, f_{i,2}, \cdots, f_{i,n_i}$ be the fractions in F_i sorted in increasing order. It's obvious that $f_{i,1} = 0$ and $f_{i,n} = 1$. Therefore, there exists an f_{i,j_i} where $1 \leq j_i \leq n_i$ such that $\mathbf{p}[i] \leq f_{i,j_i}$ and $\mathbf{p}[i] > f_{i,j_i-1}$ if $j_i \neq 1$. We set $\hat{\mathbf{p}}[i]$ to f_{i,j_i}, i.e., the smallest fraction in F_i no less than $\mathbf{p}[i]$. Let $\hat{Q}$ be the $(\mathbf{k}, \hat{\mathbf{p}})$-core. We next prove $\hat{Q} = Q$.

Apparently, we have $\mathbf{p} \leq \hat{\mathbf{p}}$. Based on Property 2, it holds that $\hat{Q} \subseteq Q$. By Definition 3.3, Q is a $\mathbf{k}[l]$-core, and for $i = 1, 2, \cdots, l-1$, there is a $\mathbf{k}[i]$-core Q_i in $G_i[Q]$ such that $\phi(v, Q_i) \geq \mathbf{p}[i]$ for all $v \in Q$. For each vertex $v \in Q$, the number of its cross-layer neighbors falling in $Q_i \subseteq V_i$ ranges from 0 to $deg_i(v)$, so $\phi(v, Q_i)$ must be in the set $\left\{0, \frac{1}{deg_i(v)}, \frac{2}{deg_i(v)}, \ldots, 1\right\} \subseteq F_i$. As $\phi(v, Q_i) \geq \mathbf{p}[i]$, we must have $\phi(v, Q_i) \geq f_{i,j_i}$. Thus, we have $\phi(v, Q_i) \geq \hat{\mathbf{p}}[i]$ for $v \in Q$. According to the definition of the $(\mathbf{k}, \hat{\mathbf{p}})$-core and the maximality of $\hat{Q}$, $Q \subseteq \hat{Q}$ holds. Overall, we have $Q = \hat{Q}$. The lemma holds. □

Proof. **(Lemma 5.3)** We first prove for each possible value of $\mathbf{k}$, there is exactly one $\mathbf{k}$-node in the KP-tree. Let $\text{sum}(\mathbf{k})$ denote the sum of all elements of vector $\mathbf{k}$. The possible values for $\mathbf{k}$ range from $\mathbf{0}^l$ to $(\kappa(G_1), \kappa(G_2), \cdots, \kappa(G_l))$. Let K be the set of all possible values of $\mathbf{k}$. We have $0 \leq \text{sum}(\mathbf{k}) \leq \sum_{i=1}^{l} \kappa(G_i)$ for any $\mathbf{k} \in K$. Thus, we only need to prove by induction that there is exactly one $\mathbf{k}$-node in the KP-tree for each possible $\mathbf{k} \in K$ sum to s for $s = 0, 1, \cdots, \sum_{i=1}^{l} \kappa(G_i)$.

Base case: When $s = 0$, $\mathbf{k} = \mathbf{0}^l$ is the only vector in K sum to s. As we have introduced in Section 5.2.1, the $\mathbf{0}^l$-node is the root of the KP-tree, which is certainly unique. Therefore, the inductive case holds.

Induction Step: Let $s^* \in \{1, 2, \cdots, \sum_{i=1}^{l} \kappa(G_i) - 1\}$ be given and suppose when $s = s^*$, there is exactly one $\mathbf{k}$-node for each possible $\mathbf{k} \in K$ sum to s. Let $\mathbf{k}'$ be any vector in K sum to $s + 1$. We construct a l-dimensional vector $\mathbf{k}''$ by decreasing the i-th element in $\mathbf{k}'$ by 1, where i is the last non-zero element in $\mathbf{k}'$. It's obvious that $\text{sum}(\mathbf{k}'') = s$. By the induction hypothesis, there is exactly one $\mathbf{k}''$-node in the KP-tree. According to Definition 5.2, $\mathbf{k}'$ is a immediate suffix successor of $\mathbf{k}''$. Therefore, the $\mathbf{k}'$-node is also in the KP-tree, specifically, is a child of the $\mathbf{k}''$-node. It's obvious that $\mathbf{k}''$ is the only vector in K of which $\mathbf{k}'$ is an immediate suffix successor. Therefore, there is exactly one $\mathbf{k}'$-node in the KP-tree, and the proof for the induction step is complete.

Conclusion: By the principle of induction, there is exactly one $\mathbf{k}$-node in the KP-tree for each possible $\mathbf{k} \in K$ sum to s for $s = 0, 1, \cdots, \sum_{i=1}^{l} \kappa(G_i)$, which certainly covers all possible values of $\mathbf{k}$.

We next prove that in every $\mathbf{k}$-node of the KP-tree, for each possible value of $\mathbf{p}$, there is exactly one $\mathbf{p}$-node in the P-tree nested in the $\mathbf{k}$-node.

As stated in Section 5.1, each element $\mathbf{p}[i]$ of $\mathbf{p}$ is chosen from the set F_i fractional numbers formulated in Equation 1. Besides, we actually store the integral form of each $\mathbf{p}$, i.e., each i-th element of $\mathbf{p}$ is replaced by its index in F_i. Therefore, the possible values of $\mathbf{p}$ range from $\mathbf{0}^{l-1}$ to $(|F_1| - 1, |F_2| - 1, \cdots, |F_{l-1}| - 1)$. Let P be the set of all possible values of $\mathbf{p}$. We have $0 \leq \text{sum}(\mathbf{p}) \leq \sum_{i=1}^{l-1}(|F_i| - 1)$ for any $\mathbf{p} \in P$, where $\text{sum}(\mathbf{p})$ is the sum of all elements of $\mathbf{p}$. Thus, we only need to prove by induction that there is exactly one $\mathbf{p}$-node in the P-tree nested in each $\mathbf{k}$-node of the KP-tree for every possible $\mathbf{p} \in P$ sum to s for $s = 0, 1, \cdots, \sum_{i=1}^{l-1}(|F_i| - 1)$.

Base case: When $s = 0$, $\mathbf{p} = \mathbf{0}^{l-1}$ is the only vector in P_{int} sum to s. As we have introduced in Section 5.2.1, the $\mathbf{0}^{l-1}$-node is the root of P-tree nested in each $\mathbf{k}$-node of the KP-tree, which is definitely unique. Therefore, the inductive case holds.

Induction Step: Let $s^* \in \{1, 2, \cdots, \sum_{i=1}^{l-1}(|F_i| - 1) - 1\}$ be given and suppose when $s = s^*$, there is exactly one $\mathbf{p}$-node in P-tree nested in each $\mathbf{k}$-node of the KP-tree for every possible $\mathbf{p} \in P$ sum to s. Let $\mathbf{p}'$ be any vector in P sum to $s + 1$. We construct a $(l-1)$-dimensional vector $\mathbf{p}''$ by decreasing the i-th element in $\mathbf{p}'$ by 1, where i is the last non-zero element in $\mathbf{p}'$. As $\text{sum}(\mathbf{p}'') = s$, we have there is exactly one $\mathbf{p}''$-node in each P-tree by the induction hypothesis. Definition 5.2 ensures that $\mathbf{p}'$ is and only is a immediate suffix successor of $\mathbf{p}''$. Thus, the $\mathbf{p}'$-node is unique in the P-tree as a child of the $\mathbf{p}''$-node. The proof for the induction step is complete.

Conclusion: By the principle of induction, there is exactly one $\mathbf{p}$-node in the P-tree nested in each $\mathbf{k}$-node of the KP-tree for every possible $\mathbf{p} \in P$ sum to s for $s = 0, 1, \cdots, \sum_{i=1}^{l-1}(|F_i| - 1)$, which covers all possible values of $\mathbf{p}$.

Overall, lemma 5.3 holds. □

Proof. **(Lemma 5.4)** Suppose that for $1 \leq i < l - 1$, Q_i is the $\mathbf{k}[i]$-core of $G_i[Q]$ and Q'_i is the $\mathbf{k}'[i]$-core of $G_i[Q']$, respectively. We have $\mathcal{M}_{\mathbf{k}}[Q] = (\mathcal{G}, C)$ is the subgraph of $\mathcal{M}$ induced by $\mathcal{Q} = \{Q_1, Q_2, \cdots, Q_{l-1}, Q\}$, and $\mathcal{M}_{\mathbf{k}'}[Q] = (\mathcal{G}', C')$ is the subgraph of

$\mathcal{M}$ induced by $Q' = \{Q'_1, Q'_2, \cdots, Q'_{l-1}, Q'\}$. As $\mathbf{k} \le \mathbf{k}'$ and $\mathbf{p} \le \mathbf{p}'$, we have $Q' \subseteq Q$ based on Property 2 and Property 3. Therefore, $G_l[Q']$ is a subgraph of $G_l[Q]$. For each $i = 1, 2, \cdots, l-1$, let Q''_i be the $\mathbf{k}[i]$-core of $G_i[Q']$. We have $Q'_i \subseteq Q''_i \subseteq Q_i$ by the hierarchy of the k-core model. Thus, $G_i[Q'_i]$ is a subgraph of $G_i[Q_i]$. For $E'_{i,j} \in C'$, as each edge $e \in E'_{i,j}$ has its endpoints in $Q'_i \subseteq Q_i$ and $Q'_j \subseteq Q_j$, respectively, e is certainly contains in $E_{i,j}$. Therefore, $E'_{i,j} \subseteq E_{i,j} \in C$. By definition of the multi-layer subgraph, $\mathcal{M}_{\mathbf{k}'}[Q']$ is a subgraph of $\mathcal{M}_{\mathbf{k}}[Q]$. The lemma holds. □

Proof. **(Lemma 7.4)** We first prove the **sufficiency**. Suppose that the P-tree is nested in the $\mathbf{k}$-node of the KP-tree. All nodes in $[N]$ correspond to the $(\mathbf{k}, \mathbf{p})$-core, say Q, and all nodes in $[N']$ correspond to the $(\mathbf{k}, \mathbf{p}')$-core, say Q'. If $\hat{\mathbf{p}} = \hat{\mathbf{p}}'$, we have $Q = Q'$ by Property 1, and therefore have $N \cong N'$. The sufficiency holds.

We next prove the **necessity**. If $N \cong N'$, N and N' correspond to the same generalized core. For any node in $N'' \in [N]$, we have $N'' \cong N \cong N'$, and hence have $N'' \in [N']$. Thus, $[N] \subseteq [N']$. Similarly, we have $[N'] \subseteq [N]$, which finally leads to $[N'] = [N]$. The maximal vector $\hat{\mathbf{p}}$ for $[N]$ is certainly equal to the maximal vector $\hat{\mathbf{p}}'$ for $[N']$. The necessity holds. The proof for Lemma 7.4 is complete. □

Proof. **(Lemma E.1)** We prove the lemma by induction on N and i. For ease of notation, we use $\mathbf{p}_N$ to denote the vector associated with N for any node N in the P-tree T.

Base case: In the first call to Procedure Forward, N is the root of the P-tree T and $i = 1$. The integral form of $\mathbf{p}_N$ corresponds to $\mathbf{0}^{l-1}$. If $\mathbf{p}_N[i] < \mathbf{p}[i]$ (the condition in line 12) is true, the **for** loop in lines 13–15 locates the child of N, say N', whose vector $\mathbf{p}_{N'}$ differs from $\mathbf{p}_N$ in the first element, and then returns N' and i. We can easily have that N' is N's rightmost child, and thereby the integral form of $\mathbf{p}_{N'}$ corresponds to $(1, 0, \cdots, 0)$. By definition of $\hat{\mathbf{p}}$, $\hat{\mathbf{p}}[i]$ is the smallest element in F_i no less than $\mathbf{p}[i]$. Thus, $\mathbf{p}_{N'}[i] \le \hat{\mathbf{p}}[i]$ holds because $\mathbf{p}_{N'}[i]$ and $\mathbf{p}_N[i]$ are adjacent in F_i. Furthermore, we have $\mathbf{p}_{N'} \le \hat{\mathbf{p}}$. According to Property 3, the $(\mathbf{k}, \hat{\mathbf{p}})$-core is a subset of the $(\mathbf{k}, \mathbf{p}_{N'})$-core, which implies that N' is on the path from the root to the $\hat{\mathbf{p}}$-node. Otherwise, we have $\mathbf{p}_N[i] \ge \mathbf{p}[i]$, and as the result, N and $i+1$ are returned in line 17. N is on the path from the root to the $\hat{\mathbf{p}}$-node as itself is the root. Besides, $\mathbf{p}_N[i]$ must be the smallest element in F_i no less than $\mathbf{p}[i]$ because the index of $\mathbf{p}_N[i]$ is 0. Overall, the base case holds.

Induction step: Suppose that when Procedure Forward is called with input N and i, N' and i' are returned and the following is ensured: (1) N' is on the path from the root to the $\hat{\mathbf{p}}$-node; (2) if $i' = i + 1$, $\mathbf{p}_N[j]$ is the smallest element in F_j that is not less than $\mathbf{p}[j]$ for $j < i$. As the $\hat{\mathbf{p}}$-node is a descendant of N', we have $\mathbf{p}_{N'} \le \hat{\mathbf{p}}$. When Procedure Forward is next called with input N' and i', if $\mathbf{p}_{N'}[i'] < \mathbf{p}[i']$ (the condition in line 12) holds, the **for** loop in lines 13–15 locates the child of N', say N'', whose vector $\mathbf{p}_{N''}$ differs from $\mathbf{p}_{N'}$ in the i'-th element, and then returns N'' and i'. Similar to the base case, N'' is the rightmost child of N'. Due to the structure of the P-tree, the integral form of $\mathbf{p}_{N''}$ is an immediate suffix successor of the integral form of $\mathbf{p}_{N'}$, which implies (1) $\mathbf{p}_{N''}[j] = \mathbf{p}_{N'}[j]$ for $j \ne i'$ and (2) $\mathbf{p}_{N'}[i']$ and $\mathbf{p}_{N''}[i']$ are adjacent in $F_{i'}$. As we have already known $\mathbf{p}_{N'}[i'] < \mathbf{p}[i']$, we certainly have $\mathbf{p}_{N''}[i'] \le \hat{\mathbf{p}}[i']$, and thereby have $\mathbf{p}_{N''} < \hat{\mathbf{p}}$. Thus, the $(\mathbf{k}, \hat{\mathbf{p}})$-core is a subset of the $(\mathbf{k}, \mathbf{p}_{N''})$-core, which ensuring that N'' is on the path from the root to the $\hat{\mathbf{p}}$-node. Otherwise, we have $\mathbf{p}_{N'}[i'] \ge \mathbf{p}[i']$. N' and $i'+1$ will be returned in line 17. According to the induction hypothesis, N' is on the path from the root to the $\hat{\mathbf{p}}$-node. If $i' = i + 1$, it holds that $\mathbf{p}_{N'}[i'] = 0$. Thus, $\mathbf{p}_{N'}[i']$ must be the smallest in element in $F_{i'}$ no less than $\mathbf{p}[i']$. Otherwise, we have $i' = i$ and $\mathbf{p}_N[i] < \mathbf{p}[i]$. In this case, $\mathbf{p}_{N'}[i']$ is also the smallest in element in $F_{i'}$ no less than $\mathbf{p}[i']$. Overall, the proof for the induction step is complete.

Conclusion: By the principle of induction, Lemma E.1 holds. □

M.3 Theorems

M.3.1 Proof for Theorem 4.1. Let us first introduce the foundation of our proof. Recall that in Appendix B, we have introduced two notations, namely Q_l^t and Q_i^t. Q_l^t denotes the value of Q_l at the beginning of the t-th running of the **repeat** loop (line 2–9) in Algorithm 1, and Q_i^t denotes the value of Q_i obtained in line 6 of Algorithm 1 during the t-th running of the **repeat** loop.

During the execution of GCS (Algorithm 1), we regard a vertex v as *qualified* w.r.t. Q_l^t if it satisfies both the Constraint (1) and the Constraint (2) in Definition 3.3, i.e., $d_{G_l[Q_l^t]}(v) \ge \mathbf{k}[l]$ and $\phi(v, Q_i^t) \ge \mathbf{p}[i]$ for $1 \le i < l - 1$. Otherwise, v is *unqualified* and will be removed later. We have the following lemma.

Lemma M.1. *Each vertex v unqualified w.r.t. Q_l^x is also unqualified w.r.t. Q_l^y for $x, y \in \mathbb{Z}$ and $x \le y \le T$, where T is the number of times the **repeat** loop (line 2–9 in Algorithm 1) runs.*

Proof. As $x \le y$, we have $Q_i^y \subseteq Q_i^x$ for $1 \le i \le l$ by Lemma B.1. If a vertex v is unqualified w.r.t. Q_l^x, it holds either (1) $d_{G_l[Q_l^x]}(v) < \mathbf{k}[l]$ or (2) $\phi(v, Q_i^x) < \mathbf{p}[i]$ for some $i \in \{1, 2, \cdots, l-2\}$. When the first case is true, we have $d_{G_l[Q_l^y]}(v) \le d_{G_l[Q_l^x]}(v) < \mathbf{k}[l]$, and thereby v is unqualified w.r.t. Q_l^y. When the second case is true, we have $\phi(v, Q_i^y) \le \phi(v, Q_i^x) < \mathbf{p}[i]$, and v is also unqualified w.r.t. Q_l^y. Overall, the lemma holds. □

With Lemma M.1, the proof for Theorem 4.1 is given below.

Proof. **(Theorem 4.1)** Algorithm 1 iteratively removes each currently unqualified vertex, until all remaining ones in Q_l become qualified. Let Q be the value of Q_l when Algorithm 1 terminates. By Lemma M.1, each removed vertex is unqualified w.r.t. Q, which ensures the maximality of Q. According to Definition 3.3, Q is the $(\mathbf{k}, \mathbf{p})$-core. Therefore, the theorem holds. □

M.3.2 Proof for Theorem 5.5.

Proof. **(Theorem 5.5)** Lemma 5.3 ensures that for each non-empty and distinct $(\mathbf{k}, \mathbf{p})$-core in $\mathcal{M}$, there is a $\mathbf{p}$-node in the P-tree nested in the $\mathbf{k}$-node of the KP-tree corresponding to it. Lemma 5.4 and Implications I1 and I2 guarantee the correctness of the computation of each $(\mathbf{k}, \mathbf{p})$-core. Overall, the theorem holds. □

M.3.3 Proof for Theorem 6.2.

Proof. **(Theorem 6.2)** Let $N_1, N_2, \cdots, N_m$ be the leftmost path starting from N to a leaf node, where $N_1 = N$ and N_m is a leaf node. Suppose that the generalized core represented by N_i is Q_i for $1 \le i \le m$. It's obvious that $Q_m = \emptyset$. We use V_i to denote the set of vertices associated with the edge connecting N_i and N_{i+1}. By the augmentations to the P-tree that we have introduced in Section 6.1, all sets V_i are stored, and $V_i = Q_i - Q_{i+1}$. The union of the sets associated with all edges on the path, therefore, is: $\sum_{i=1}^{m-1} |V_i| = \sum_{i=1}^{m-1}(Q_i - Q_{i+1}) = Q_1$. Thus, the theorem holds. □

M.3.4 Proof for Theorem 6.3.

Proof. **(Theorem 6.3)** Line 5 finds the $\mathbf{k}$-node in the KP-tree through a hash table lookup. Lemma E.1 ensures Procedure Search (line 3) returns the $\hat{\mathbf{p}}$-node, say N, in the P-tree nested in the $\mathbf{k}$-node. Theorem 6.2 guarantees Procedure Recover (line 4) correctly retrieves the generalized core corresponding to N, which is identical to $(\mathbf{k}, \mathbf{p})$-core. Overall, the theorem holds. □

M.3.5 Proof for Theorem 7.2.

Proof. **(Theorem 7.2)** We prove the theorem by contradiction. Given P-tree T nested in the $\mathbf{k}$-node of the KP-tree, suppose that $[N]$, where N is any node in T, has two distinct maximal vectors $\mathbf{p}_1$ and $\mathbf{p}_2$. Due to the maximality of $\mathbf{p}_1$ and $\mathbf{p}_2$, there must exist two distinct integers $j_1, j_2 \in \{1, 2, \cdots, l-1\}$ such that $\mathbf{p}_1[j_1] > \mathbf{p}_2[j_1]$ and $\mathbf{p}_2[j_2] > \mathbf{p}_1[j_2]$. Let Q be the generalized core corresponding to nodes in $[N]$. By Definition 3.3, for $i = 1, 2, \cdots, l-1$, there exists a nonempty $\mathbf{k}[i]$-core Q_i in $G_i[Q]$ such that $\phi(v, Q_i) \geq \mathbf{p}_1[i]$ and $\phi(v, Q_i) \geq \mathbf{p}_2[i]$ for $v \in Q$. Let $\mathbf{p}$ be the element-wise maximum of $\mathbf{p}_1$ and $\mathbf{p}_2$, i.e., $\mathbf{p}[i] = \max\{\mathbf{p}_1[j], \mathbf{p}_2[j]\}$ for $1 \leq j \leq l-1$, and N^* be the node in T with which $\mathbf{p}$ is associated. It's obvious that $\phi(v, Q_i) \geq \mathbf{p}[i]$ for $v \in Q$, making Q also a $(\mathbf{k}, \mathbf{p})$-core. Hence, we have $N^* \in [N]$, and the fact that $\mathbf{p}'$ dominates both $\mathbf{p}_1$ and $\mathbf{p}_2$ contradicts with the maximality of $\mathbf{p}_1$ and $\mathbf{p}_2$. The theorem thus holds. □

M.3.6 Proof for Theorem 7.3.

Proof. **(Theorem 7.3)** We prove the theorem by contradiction. For ease of notation, let $\phi_i = \min_{v \in Q} \phi(v, Q_i)$, where Q_i is the $\mathbf{k}[i]$-core of $G_i[Q]$. Suppose that $\hat{\mathbf{p}}[i] \neq \phi_i$ for some $i \in \{1, 2, \cdots, l-1\}$. If $\hat{\mathbf{p}}[i] < \phi_i$, we can construct a $(l-1)$-dimensional vector $\mathbf{p}'$ by replacing the i-th element of $\mathbf{p}$ with ϕ_j. As Q is a $(\mathbf{k}, \mathbf{p})$-core, there must exists a nonempty $\mathbf{k}[i]$-core Q_i of $G_i[Q]$ such that $\phi(v, Q_i) \geq \mathbf{p}[i]$ according to Definition 3.3. It hence also holds that $\phi(v, Q_i) \geq \mathbf{p}'[i]$ for $1 \leq i \leq l-1$ since $\mathbf{p}'[i] = \max\{\mathbf{p}[i], \phi_i\}$. Q therefore is also a $(\mathbf{k}, \mathbf{p}')$-core. Let N' be the node in T with which $\mathbf{p}'$ is associated with. We have $N' \in [N]$, and the fact that $\mathbf{p}'$ dominates $\mathbf{p}$ contradicts with the maximality of $\mathbf{p}$. Otherwise, we have $\hat{\mathbf{p}}[i] > \phi_i$. According to Definition 3.3, it holds that $\min_{v \in Q} \phi(v, Q_i) \geq \mathbf{p}[i] > \phi_i$, which is also a contradiction. Overall, the theorem holds. □

M.3.7 Proof for Theorem 7.5. Let us first introduce the foundation of our proof. As node elimination is a special case of subtree elimination, we prove Theorem 7.5 holds after replacing the "node elimination" in the theorem with "subtree elimination".

Let T and T' denote any P-tree and the P+-tree obtained by repeatedly applying the subtree elimination procedure. When a $(\mathbf{k}, \mathbf{p})$-core represented by a node N in T is queried using Algorithm 3, Procedure Search locates N by traversing a series of nodes in T and comparing their vectors against $\mathbf{p}$. Let $S = N_1, N_2, \cdots, N_l$ be the sequence of nodes returned at line 17 during the execution of Search. Clearly, N_l is N. However, when using Algorithm 3 to query the $(\mathbf{k}, \mathbf{p})$-core in T', as some nodes in T are removed by the subtree elimination procedure, and thereby no longer exist in T', we suppose $S' = N'_1, N'_2, \cdots, N'_l$ are the sequence of nodes returned at line 17 instead, where $N'_l = N^*$. The following Lemma M.2 builds a bridge between S and S', which serves as a key to prove $N \cong N^*$.

Lemma M.2. *The preceding subtree rooted at N_i is isomorphic to the preceding subtree rooted at N'_i for $i = 1, 2, \cdots, l$.*

Proof. We prove the lemma by induction on i.

Base case: The inductive base is for $i = 1$. When Algorithm 3 runs on T and T', before N_1 and N'_1 are returned, it repeatedly calls Procedure Forward in line 9 to check the node under investigation, say R and R', and visit the rightmost child of R and R', respectively. N_1 is the first node visited in T whose vector has the i-th element no smaller than $\mathbf{p}[i]$, and N'_1 is the first node visited in T' whose vector has the i-th element no smaller than $\mathbf{p}[i]$. Let $P_0 = N_{0,0}, N_{0,1}, \cdots$ be the rightmost path of the root node of T, where $N_{0,0}$ is T's root, and let $P'_0 = N'_{0,0}, N'_{0,1}, \cdots$ be the rightmost path of the root node of T', where $N'_{0,0}$ is the root of T'. Obviously, there exists a node $N_{0,j}$ such that $N_{0,j} = N_1$. If $N_{0,j}$ is not eliminated and remains in P'_0, it must also be returned as N'_1 in line 17. The preceding subtrees rooted at N_1 and N'_1 are thereby isomorphic. Otherwise, suppose $N_{0,j}, N_{0,j+1}, \cdots, N_{0,k-1}$ is the continuous subsequence of P_0 starting from $N_{0,j}$ being eliminated. Certainly, $N_{0,k}$ will be returned as N'_1. As the premises of subtree elimination, the preceding subtrees rooted at each node in $\{N_{0,j}, N_{0,j+1}, \cdots, N_{0,k-1}\}$ are isomorphic and are isomorphic to the subtree rooted at $N_{0,k}$. Therefore, the preceding subtrees rooted at N_1 and N'_1 are also isomorphic in this case. Overall, the base case holds.

Induction step: Let $i^* \in \{1, 2, \cdots, l-1\}$ be given and suppose when $i = i^*$, the preceding subtree rooted at N_i is isomorphic to the preceding subtree rooted at N'_i. When N_i and N'_i are returned at line 17, Algorithm 3 continues calling Procedure Forward to visit and check the nodes on the rightmost path of N_i and N'_i, respectively. The process repeats until the first nodes in P_i and P'_i with the $(i+1)$-th element of their vectors no smaller than $\mathbf{p}[i+1]$ are visited, where P_i is the rightmost path of N_i in T and P'_i is the rightmost path of N'_i in T. Then, such nodes will be returned as N_{i+1} and N'_{i+1} in line 17, respectively. Suppose $P_i = N_{i,0}, N_{i,1}, \cdots$, where $N_{i,0} = N_i$, and $P'_i = N'_{i,0}, N'_{i,1}, \cdots$, where $N'_{i,0} = N'_i$. The induction hypothesis ensures the preceding subtrees rooted at N_i and N'_i are isomorphic. By Definition 7.8, we can derive that the preceding subtrees rooted at each pair of $N_{i,j}$ and $N'_{i,j}$ are isomorphic for $j \in \mathbb{N}$. Besides, the structure of the P-tree ensures the $(i+1)$-th elements in each pair of $N_{i,j}$ and $N'_{i,j}$ are identical. Similar to the base case, there must be a node $N_{i,j}$ in P_i that is N_{i+1}. If $N'_{i,j}$ is not removed by the subtree elimination procedure, it will definitely be returned as N'_{i+1} because $\mathbf{p}'_{i,j}[i] = \mathbf{p}_{i,j}[i]$, where $\mathbf{p}_{i,j}$ and $\mathbf{p}'_{i,j}$ are the vectors associated with $N_{i,j}$ and $N'_{i,j}$, respectively. In this case, the isomorphism between the preceding subtrees rooted at N_{i+1} and N'_{i+1} is clear. Otherwise, let $N'_{1,j}, N'_{1,j+1}, \cdots, N'_{1,k-1}$ be the continuous subsequence of P'_i starting from $N'_{1,j}$ being eliminated. Obviously, $N'_{1,k} \in P'_i$ will be returned as N'_{i+1}. As the premise of the subtree elimination, the preceding subtrees rooted at all $N'_{1,j'}$ for $j \leq j' \leq k$ are isomorphic. Thus, by the transitivity of $\cong$, we have the preceding subtrees rooted at N_{i+1} and N'_{i+1} are isomorphic. The proof for the induction step is complete.

Conclusion: By the principle of induction, the preceding subtrees rooted at each pair of N_i and N'_i are isomorphic, where $1 \leq i \leq l$. □

With Lemma M.2, the proof for Theorem 7.5 is presented below.

Proof. **(Theorem 7.5)** By Lemma M.2, the preceding subtrees rooted at N_l and N'_l are isomorphic, which certainly implies $N_l \cong N'_l$, i.e., $N \cong N^*$. Let $P = R_0, R_1, \cdots$ be nodes on the leftmost path starting from N' in T, where $R_0 = N'$. It's clear that R_j has only one child R_{j+1} for $j \in \mathbb{Z}$. If some $R_j \in P$ is removed, where $j \in \mathbb{Z}$, $R_j \cong R_{j+1}$ must hold, and the vertex set associated with the edge connecting R and R' is empty. In other words, applying subtree elimination to nodes in P involves only removing edges with an empty vertex set on the leftmost path of N'. Thus, all nonempty vertex sets on the leftmost path of N' in T are retained on the leftmost path of N^* in T'. Overall, the theorem holds. □

M.3.8 Proof for Theorem 7.6. Let us first introduce the foundation of our proof. The following Lemma M.3 gives a sufficient condition to determine the redundancy between any two nodes in a P-tree.

Lemma M.3. *Given any two nodes N and N' in a P-tree, let $\mathbf{p}$ and $\mathbf{p}'$ be the vectors associated with N and N', respectively, and let $\hat{\mathbf{p}}$ and $\hat{\mathbf{p}}'$ be the maximal vectors for $[N]$ and $[N']$, respectively. We have $N \cong N'$ if either $\mathbf{p} \leq \mathbf{p}' \leq \hat{\mathbf{p}}$ or $\mathbf{p}' \leq \mathbf{p} \leq \hat{\mathbf{p}}'$ holds.*

Proof. **(Lemma M.3)** Without loss of generality, we suppose $\mathbf{p} \leq \mathbf{p}' \leq \hat{\mathbf{p}}$ holds. Let Q be the generalized core corresponding to nodes in $[N]$, and Q' be the generalized core corresponding to nodes in $[N']$. By Property 3, the left and right inequalities imply $Q' \subseteq Q$ and $Q \subseteq Q'$, respectively. Thus, we have $Q = Q'$. The lemma holds. □

With Lemma M.3, the proof for Theorem 7.6 is given below.

Proof. **(Theorem 7.6)** Let $\mathbf{p}$ be the vector associated with N. We first prove the sufficiency. As N' is a child of N, we have $\mathbf{p} < \mathbf{p}'$. Together with the condition that $\mathbf{p}' \leq \hat{\mathbf{p}}$, we have $N \cong N'$ by Lemma M.3. The sufficiency holds.

We next prove the **necessity**. If $N \cong N'$, we have $N' \in [N]$. By Definition 7.1, $\mathbf{p}'' \leq \hat{\mathbf{p}}$ for all vectors $\mathbf{p}''$ associated with nodes in $[N]$. Thus, it certainly holds that $\mathbf{p}' \leq \hat{\mathbf{p}}$. The necessity thereby holds. Overall, the proof for Theorem 7.6 is complete. □

M.3.9 Proof for Theorem 7.10. Let us first introduce the foundation of our proof. For ease of notation, let $sd(\mathbf{p})$ denote the smallest dimension of a vector $\mathbf{p}$ at which $\mathbf{p}$ differs from its immediate suffix successors. According to Definition 5.2, it can be easily derived that $\mathbf{p}[i] = 0$ for all $i > sd(\mathbf{p})$. We then present several observations on the structures of P-trees.

Observation 1. *Given any node N in a P-tree, let $\mathbf{p}$ be the vector associated with N. We have that N has $l - sd(\mathbf{p})$ children, and for $i = sd(\mathbf{p}), sd(\mathbf{p}) + 1, \cdots, l - 1$, there is a child N' of N such that the vector associated with N' differs from $\mathbf{p}$ in the i-th element.*

Observation 2. *Given any node N in a P-tree, let $\mathbf{p}$ be the vector associated with N, and let T be the subtree rooted at N. N' is a node in T if and only if $\mathbf{p}[i] = \mathbf{p}'[i]$, where $\mathbf{p}'$ for $i = 1, 2, \cdots, sd(\mathbf{p}) - 1$ where $\mathbf{p}'$ is the vector associated with N', i.e., $\mathbf{p}$ and $\mathbf{p}'$ share a prefix of length $sd(\mathbf{p}) - 1$.*

Based on the above observations, we introduce the following two lemmas. Lemma M.4 gives a sufficient condition to determine the isomorphism between any two subtrees in a P-tree, and Lemma M.5 gives a sufficient condition to determine the isomorphism between any two preceding subtrees in a P-tree.

Lemma M.4. *Given any two nodes N_1 and N_2 in a P-tree, let $\mathbf{p}_1$ and $\mathbf{p}_2$ be the vectors associated with N_1 and N_2, respectively, and let $\hat{\mathbf{p}}_1$ and $\hat{\mathbf{p}}_2$ be the signature of the subtree rooted at N_1 and N_2, say T_1 and T_2, respectively. We have $T_1 \cong T_2$ if (1) $\#sd(\mathbf{p}_1) = \#sd(\mathbf{p}_2)$, say i; (2) $\mathbf{p}_1[i] = \mathbf{p}_2[i]$; and (3) either $\mathbf{p}_1[j] \leq \mathbf{p}_2[j] \leq \hat{\mathbf{p}}_1[j]$ for $1 \leq j \leq i-1$ or $\mathbf{p}_2[j] \leq \mathbf{p}_1[j] \leq \hat{\mathbf{p}}_2[j]$ for $1 \leq j \leq i-1$ hold.*

Proof. **(Lemma M.4)** Let h be the height of T_1, we next prove the lemma by induction on h. Obviously, h ranges from 1 to h_T, where h_T is the height of the P-tree T.

Base case: When $h = 1$, T_1 consists of a single root node N_1. Obviously, N_1 is a leaf node of T and thereby represents an empty gCore. Without loss of generality, suppose $\mathbf{p}_1[j] \leq \mathbf{p}_2[j] \leq \hat{\mathbf{p}}_1[j]$ for $1 \leq j \leq i-1$ in Condition (3) holds. Based on Condition (1), we have $\mathbf{p}_1[j] = \mathbf{p}_2[j] = 0 \leq \hat{\mathbf{p}}_1[j]$ for $j > i$. By Condition (2), we have $\mathbf{p}_1[i] = \mathbf{p}_2[i] \leq \hat{\mathbf{p}}_1[i]$. Thus, $\mathbf{p}_1 \leq \mathbf{p}_2 \leq \hat{\mathbf{p}}_1$, and $N_1 \cong N_2$ by Lemma M.3. Thus, N_2 also represents an empty gCore, and serves as a leaf node of T. T_2 thereby consists of N_2 only. By Definition 7.8, T_1 is isomorphism to T_2 because $N_1 \cong N_2$. The base case holds.

Induction step: Let $h^* \in \{2, 3, \cdots, h_T - 1\}$ be given and suppose Lemma M.4 holds if the height of T is h, where $1 \leq h \leq h^*$. Assume T_1 rooted at N_1 is a subtree of height $h+1$ in T. As we have proved in the base case, $N_1 \cong N_2$. According to Condition (1) and Observation 1, N_1 and N_2 have the same number of children. Let N'_1 be the child of N_1 with vector $\mathbf{p}'_1$ different from $\mathbf{p}_1$ in the j-th element, and N'_2 be the child of N_2 with vector $\mathbf{p}'_2$ different from $\mathbf{p}_2$ in the j-th element, where $i \leq j \leq l-1$. We use T'_1 and T'_2 to denote the subtrees rooted at N'_1 and N'_2, respectively. It's obvious that $sd(\mathbf{p}'_1) = sd(\mathbf{p}'_2) = j$. If $j = i$, $\mathbf{p}'_1[j] = \mathbf{p}'_2[j] = \mathbf{p}_1[j] + 1$. Otherwise, we have $j > i$, and hence $\mathbf{p}'_1[j] = \mathbf{p}'_2[j] = 1$. Therefore, $\mathbf{p}'_1[j] = \mathbf{p}'_2[j]$ always holds. Without loss of generality, we suppose that $\mathbf{p}_1[j] \leq \mathbf{p}_2[j] \leq \hat{\mathbf{p}}_1[j]$ for $1 \leq j \leq i-1$ in Condition (3) holds. The left inequality implies $\mathbf{p}'_1[k] \leq \mathbf{p}'_2[k]$ for $1 \leq k \leq i-1$. For $i \leq k < j$, we have $\mathbf{p}'_1[k] = \mathbf{p}'_2[k] = 0$. Thus, $\mathbf{p}'_1[k] \leq \mathbf{p}'_2[k]$ for all $1 \leq k \leq j-1$. Let $\hat{\mathbf{p}}'_1$ be the signature of T'_1. As all nodes in T'_1 must be in T_1, by Definition 7.9, we have $\hat{\mathbf{p}}_1 \leq \hat{\mathbf{p}}'_1$. With the right inequality in the above Condition (3), it holds for $1 \leq k < i$ that $\mathbf{p}'_2[k] = \mathbf{p}_2[k] \leq \hat{\mathbf{p}}'_1[k]$. For $i \leq k < j$, we have $\mathbf{p}'_2[k] = 0 \leq \hat{\mathbf{p}}'_1[k]$. Thus, $\mathbf{p}'_2[k] \leq \hat{\mathbf{p}}'_1[k]$ for all $1 \leq k \leq j-1$. Overall, we have $\mathbf{p}'_1[k] \leq \mathbf{p}'_2[k] \leq \hat{\mathbf{p}}'_1[k]$ for each $k < j$. As T'_1 and T'_2 must have heights no larger than h, the induction hypothesis ensures that $T'_1 \cong T'_2$. Notice N'_1 and N'_2 are the $(j - i + 1)$-th child of N_1 and N_2 in sorted order of $\prec$ (Definition 6.1), respectively. Due to the arbitrariness of j and Definition 7.8, T_1 and T_2, both with height $h+1$, are isomorphic. The proof for the induction step is complete.

Conclusion: By the principle of induction, Lemma M.4 holds for T_1 with height ranging from 1 to h_T, which certainly covers all subtrees in T. Overall, the lemma holds. □

Lemma M.5. *Given any two nodes N_1 and N_2 in a P-tree, let $\mathbf{p}_1$ and $\mathbf{p}_2$ be the vectors associated with N_1 and N_2, respectively, let $\hat{\mathbf{p}}_1$ and $\hat{\mathbf{p}}_2$ be the maximal vectors for $[N_1]$ and $[N_2]$, respectively, and let $\hat{\mathbf{p}}'_1$ and $\hat{\mathbf{p}}'_2$ be the signature of the preceding subtree rooted at N_1 and N_2, say T_1 and T_2, respectively. We have $T_1 \cong T_2$ if (1) $sd(\mathbf{p}_1) = sd(\mathbf{p}_2)$, say i; and (2) either $\mathbf{p}_1[i] \le \mathbf{p}_2[i] \le \hat{\mathbf{p}}_1[i]$ and $\mathbf{p}_1[j] \le \mathbf{p}_2[j] \le \hat{\mathbf{p}}'_1[j]$ for $1 \le j < i$ or $\mathbf{p}_2[i] \le \mathbf{p}_1[i] \le \hat{\mathbf{p}}_2[i]$ and $\mathbf{p}_2[j] \le \mathbf{p}_1[j] \le \hat{\mathbf{p}}'_2[j]$ for $1 \le j < i$ hold.*

Proof. **(Lemma M.5)** According to Condition (1) and Observation 1, $\mathbf{p}_1[j] = \mathbf{p}_2[j] = 0$ for $j > i$. Thus, we have $\mathbf{p}_1[j] \le \mathbf{p}_2[j] \le \hat{\mathbf{p}}_1[j]$ for $i < j \le l-1$. Without loss of generality, suppose that $\mathbf{p}_1[i] \le \mathbf{p}_2[i] \le \hat{\mathbf{p}}_1[i]$ and $\mathbf{p}_1[j] \le \mathbf{p}_2[j] \le \hat{\mathbf{p}}'_1[j]$ for $1 \le j < i$ in Condition (2) are true. As N_1 is contained in T_1, we have $\hat{\mathbf{p}}'_1 \le \hat{\mathbf{p}}_1$ by Definition 7.1. Thus, $\mathbf{p}_1[j] \le \mathbf{p}_2[j] \le \hat{\mathbf{p}}_1[j]$ for $1 \le j \le i$, and moreover, $\mathbf{p}_1 \le \mathbf{p}_2 \le \hat{\mathbf{p}}_1$. By Lemma M.3, $N_1 \cong N_2$.

Obviously, N_1 and N_2 have the same number of children in T_1 and T_2, respectively. Let N'_1 be the child of N_1 with vector $\mathbf{p}'_1$ different from $\mathbf{p}_1$ in the j-th element, and N'_2 be the child of N_2 with vector $\mathbf{p}'_2$ different from $\mathbf{p}_2$ in the j-th element, where $i < j \le l-1$. And let T'_1 and T'_2 be the subtrees rooted at N'_1 and N'_2, respectively. We have $sd(\mathbf{p}'_1) = sd(\mathbf{p}'_2) = j$, and $\mathbf{p}'_1[j] = \mathbf{p}'_2[j] = 1$. As proved before, $\mathbf{p}_1 \le \mathbf{p}_2$, we have $\mathbf{p}'_1[k] \le \mathbf{p}'_2[k]$ for $k < j$. Let $\hat{\mathbf{p}}'_1$ be the signature of T'_1. All nodes in T'_1 must be in T_1, $\hat{\mathbf{p}}_1 \le \hat{\mathbf{p}}'_1$ is thus ensured by Definition 7.1. Then based on Condition (2), we have $\mathbf{p}'_2[k] = \mathbf{p}_2[k] \le \hat{\mathbf{p}}'_1[k]$ for $1 \le k < i$, and $\mathbf{p}'_2[k] = 0 \le \hat{\mathbf{p}}'_1[k]$ for $1 \le k < j$. That is, $\mathbf{p}'_2[k] \le \hat{\mathbf{p}}'_1[k]$ for all $k < j$. Overall, $\mathbf{p}_1[k] \le \mathbf{p}'_2[k] \le \hat{\mathbf{p}}'_1[k]$ is true for $1 \le k < j$. According to Lemma M.4, T'_1 is isomorphic to T'_2, i.e., $T'_1 \cong T'_2$. Notice N'_1 and N'_2 are the $(j-i+1)$-th child of N_1 and N_2 in sorted order of $\prec$ (Definition 6.1), respectively. Due to the arbitrariness of j and Definition 7.8, we have $T_1 \cong T_2$. The proof for Lemma M.5 is thereby complete. □

With the above observations and lemmas, the proof for Theorem 7.10 is given below.

Proof. **(Theorem 7.10)** We first prove the **sufficiency**. As N' is a descendant of N on N's rightmost path, we have $\mathbf{p} < \mathbf{p}'$ and $sd(\mathbf{p}) = sd(\mathbf{p}') = i$. If the condition $\mathbf{p}'[i] < \hat{\mathbf{p}}[i]$ is true, it holds that $\mathbf{p}[i] \le \mathbf{p}'[i] < \hat{\mathbf{p}}[i]$. According to Observation 2, $\mathbf{p}$ and $\mathbf{p}'$ share a prefix of length $i-1$. We then have $\mathbf{p}'[j] = \mathbf{p}[j] \le \hat{\mathbf{p}}[j]$ for $1 \le j \le i-1$. Thus, $\mathbf{p}[j] \le \mathbf{p}'[j] \le \hat{\mathbf{p}}[j]$ holds for each $j < i$. By Lemma M.5, T is isomorphic to T'. The sufficiency thereby holds.

We next prove the **necessity**. If $T \cong T'$, for each node R' in T', there is a node R in T such that $R' \cong R$. Therefore, according to Definition 7.9, we have $\hat{\mathbf{p}}' = \hat{\mathbf{p}}$, where $\hat{\mathbf{p}}'$ is the signature of T'. For each node R' in T' with vector $\mathbf{p}''$, R' is a either N' or a descendant of N'. It thus holds that $\mathbf{p}' \le \mathbf{p}'' \le \hat{\mathbf{p}}''$, where $\hat{\mathbf{p}}''$ is the maximal vector for $[R']$. Also by Definition 7.9, we have $\mathbf{p}' \le \hat{\mathbf{p}}'$. Overall, $\mathbf{p}' \le \hat{\mathbf{p}}$, which certainly implies that $\mathbf{p}'[i] \le \hat{\mathbf{p}}[i]$. The necessity holds, and the proof for Theorem 7.10 is complete. □

M.3.10 Proof for Theorem 7.12. Let us first introduce the foundation of our proof. We use T to denote any P+-tree and T' to denote the P+-DAG obtained by repeatedly applying the subtree merge procedure to T. Let T^c be the original P-tree of T. We suppose T is obtained by repeatedly applying the subtree elimination procedure to T^c until no more nodes can be removed. The assumption is rational because (1) as we have stated in Section 7.5, to make T' more compact, we apply subtree merge after doing a sequence of subtree elimination operation; (2) such assumption simplifies the proof for Theorem 7.12. In fact, Theorem 7.12 holds for any P+-tree, which can be proved by a more complicated induction.

When the $(\mathbf{k}, \mathbf{p})$-core represented by a node N in T is queried using Algorithm 3, Procedure Search repeatedly calls Procedure Forward to traverse a series of nodes in T and compare their vectors against $\mathbf{p}$. The process repeats until N has been found. Let $O = O_1, O_2, \cdots, O_n$ be the sequence of outputs of Forward at either line 15 or line 17 during the execution of Search, where each O_j with $1 \le j \le n$ is the pair (N_j, i_j) returned at the j-th call to Forward. We have $O_n = (N, l)$. After applying the subtree merge procedure to T, some subtrees in T may be removed and replaced by other subtrees. Thus, using Algorithm 3 to search the $(\mathbf{k}, \mathbf{p})$-core on T' may lead to another sequence of outputs of Procedure Forward, say $O' = O'_1, O'_2, \cdots, O'_{n'}$, where $O'_j = (N'_j, i'_j)$ for $1 \le j \le n'$ and $O'_{n'} = (N^*, l)$. The following lemma builds a bridge between O and O', which is key to the proof for $N \cong N^*$.

Lemma M.6. *It holds for $j = 1, 2, \cdots$ that (1) either $\min\{n, n'\} > j$ or $n = n' = j$; (2) $i_j = i'_j$, say i; (3) $\mathbf{p}_j[i] = \mathbf{p}'_j[i]$, where $\mathbf{p}_j$ and $\mathbf{p}'_j$ are vectors associated with N_j and N'_j, respectively, and (4) the subtree rooted at N_j in T^c is isomorphic to the subtree rooted at N'_j in T^c.*

To facilitate the proof for Lemma M.6, Lemma M.7 is given next.

Lemma M.7. *Given a P-tree T and a P+-tree T' obtained by repeatedly applying the subtree elimination procedure on T until no more nodes can be removed. For any two nodes N_1 and N_2 in T, let $\mathbf{p}_1$ and $\mathbf{p}_2$ be the vectors associated with N_1 and N_2, respectively. If the subtree rooted at N_1 is isomorphic to the subtree rooted at N_2 in T and $\mathbf{p}_1[i] = \mathbf{p}_2[i]$, where $i = sd(\mathbf{p}_1)$, we have the subtree rooted at N'_1 is isomorphic to the subtree rooted at N'_2 in T and $\mathbf{p}'_1[j] = \mathbf{p}'_2[j]$, where N'_1 is a child of N_1 whose vector $\mathbf{p}'_1$ differs from $\mathbf{p}_1$ in the j-th element in T', and N'_2 is a child of N_2 whose vector $\mathbf{p}'_2$ differs from $\mathbf{p}_2$ in the j-th element in T'.*

Proof. **(Lemma M.7)** Let us first introduce some notations. We use R_1 and R_2 to denote the children of N_1 and N_2 in T with vectors differ from $\mathbf{p}_1$ and $\mathbf{p}_2$ in the j-th element, respectively. Let $P_1 = R_{1,0}, R_{1,1}, \cdots, R_{1,n_1}$ be the rightmost path starting from R_1 in T, where $R_{1,0} = R_1$, and let $P_2 = R_{2,0}, R_{2,1}, \cdots, R_{2,n_2}$ be the rightmost path starting from R_2 in T, where $R_{2,0} = R_2$. We use $\mathbf{p}_{1,k_1}$ and $\mathbf{p}_{2,k_2}$ denote the vectors associated with R_{1,k_1} and R_{2,k_2}, respectively, where $0 \le k_1 \le n_1$ and $0 \le k_2 \le n_2$.

By Definition 7.8, the isomorphism between subtrees rooted at N_1 and N_2 in T ensures $n_1 = n_2$, say n, and the isomorphism between subtrees rooted at each pair of $R_{1,k}$ and $R_{2,k}$ in T for $0 \le k \le n$. Besides, according to the structure of the P-tree, the j-th elements of vectors of nodes in P_1 and P_2 are monotonically increasing, holding that $\mathbf{p}'[j] = \mathbf{p}[j] + 1$ for any pair of adjacent nodes R and R' with vectors $\mathbf{p}$ and $\mathbf{p}'$ in P_1 or P_2, respectively. As $\mathbf{p}_1[i] = \mathbf{p}_2[i]$ and $j \ge i$, we have $\mathbf{p}_{1,j}[j] = \mathbf{p}_{2,j}[j] = p$, where $p = \mathbf{p}_1[i] + j + 1$ when $j = i$, and $p = j + 1$ when $j > i$. Thus, the j-th elements of each pair of $R_{1,k}$ and $R_{2,k}$ are always identical. Obviously, there exist two integers k_1 and k_2, where $0 \leq k_1, k_2 \leq n$, such that $R_{1,k_1} = N_1'$ and $R_{2,k_2} = N_2'$. To prove the isomorphism between subtrees rooted at N_1' and N_2' and the identity between the j-th elements in their vectors, we only need to prove $k_1 = k_2$.

Next, we prove $k_1 = k_2$ by a simple contradiction. If $k_1 \neq k_2$, without loss of generality, we suppose $k_1 < k_2$. Let R be the node R_{2,k_1}. Clearly, R is in the sequence $P_2' = R_{2,0}, R_{2,1}, \cdots, R_{2,k_2-1}$. Let us consider two nodes R_{1,k_1+1} and R_{2,k_1+1}, the child node of R_{1,k_1} and R, respectively. We have (1) $\mathbf{p}_{1,k_1+1}[j] = \mathbf{p}_{2,k_1+1}[j]$ (2) the subtrees rooted at R_{1,k_1+1} and R_{1,k_2+1} are isomorphic. Certainly, the isomorphism between the subtrees rooted at R_{1,k_1} and R ensures the isomorphism between the preceding subtrees rooted at R_{1,k_1} and R, and the isomorphism between the subtrees rooted at R_{1,k_1+1} and R_{1,k_2+1} ensures the isomorphism between the preceding subtrees rooted at them. As all nodes in P_2' are removed by the subtree elimination procedure, as the premises of which, the preceding subtrees rooted at all nodes in P_2' are isomorphic, and are isomorphic to the preceding subtree rooted at R_{2,k_2}. It's obvious that R_{2,k_1+1} is either in P_2' or exactly is R_{2,k_2}. We hence have the preceding subtrees rooted at R and R_{2,k_1+1} are isomorphic. By transitivity of $\cong$, the preceding subtrees rooted at R_{1,k_1} and R_{1,k_1+1} are isomorphic. R_{1,k_1}, i.e., N_1' will therefore be removed by the subtree elimination procedure. It contradicts with the hypothesis that no nodes can be removed from T'. Overall, we have $k_1 = k_2$, and hence the proof for the lemma is complete. □

With Lemma M.7, the proof for Lemma M.6 is given as below.

Proof. **(Lemma M.6)** We prove the lemma by induction on j. As the terminal condition at line 8 in Search is reached when i is increased to l from 1, and each increase of i corresponds to a call to Forward and return at line 17, both n and n' are no less than $l - 1$. Without loss of generality, we suppose $l \geq 2$.

Base case: The inductive base is for $j = 1$. Let R and R' be the root of T and T', respectively, and $\mathbf{p}_0$ and $\mathbf{p}_0'$ be the vector associated with R and R', respectively. The first call to Procedure Forward on T is given R and 1 as input, and the call on T' is given R' and 1 as input. Apparently, the subtrees rooted at R and all nodes on the rightmost path starting from R cannot be removed by the subtree merge procedure. We, therefore, have $R = R'$, and the rightmost child of R is identical to the rightmost child of R', say R''. If $\mathbf{p}_0[i] < \mathbf{p}[i]$, the rightmost child of R and R' will be returned at line 15. We then have $N_j = N_j' = R''$ and $i_j = i_j' = 1$. The isomorphism between subtrees rooted at N_j and N_j' in T^c is obvious, and $\mathbf{p}_j[i] = \mathbf{p}_j'[i]$. Besides, both n and n' are larger than j since $i < l$.

Otherwise, we have $\mathbf{p}_0[i] \geq \mathbf{p}[i]$. In this case, R and R' will be returned at line 17. Thus, it holds $N_j = N_j' = R$ and $i_j = i_j' = 2$. The isomorphism between subtrees rooted at N_j and N_j' in T^c is also obvious. And the i-th elements of $\mathbf{p}_j$ and $\mathbf{p}_j'$ are identical. If $l > 2$, Procedure Forward will be continually called to test new nodes, and both n and n' are larger than j. Otherwise, we have $l = 2$. The **while** loop in lines 8–9 terminates, and N_j and N_j' are returned as N and N^* at line 10. It holds that $n = n' = j$. Overall, the base case holds.

Inductive step: Let $j^* \in \mathbb{Z}$ be given and suppose when $j = j^*$, it holds $j < \min\{n, n'\}$, $i_j = i_j'$, $\mathbf{p}_j[i] = \mathbf{p}_j'[i]$ with $i = i_j$, and the subtree rooted at N_j in T^c is isomorphic to the subtree rooted at N_j' in T^c. In the $(j+1)$-th call to Procedure Forward on T and T', the input are (N_j, i) and (N_j', i), respectively. Let R be the child of N_j with vector different from $\mathbf{p}_j$ only in the i-th element in T, and R' be the child of N_j' with vector different from $\mathbf{p}_j'$ in the i-th element in T. Notice that the vector associated with R' may differ from $\mathbf{p}_j'$ in multiple elements because of subtree merge, while it ensures that they are different only in the first i-th elements. There are two cases:

(1) If $\mathbf{p}_j[i] \leq \mathbf{p}[i]$, (R, i) and (R', i) will be returned at line 15. It holds $i_{j+1} = i_{j+1}' = i$. Besides, $i < l$ definitely holds, otherwise Procedure Search terminates before $(j+1)$-th call to Procedure Forward. Thus, we have $\min\{n, n'\} > j$. Let R'' be the child of N_j' in T with vector only different from $\mathbf{p}_j'$ in the j-th element. By induction hypothesis, the subtrees rooted at N_j and N_j' in T^c are isomorphic and $\mathbf{p}_j[i] = \mathbf{p}_j'[i]$ holds. According to Lemma M.7, we have the subtrees rooted at R and R'' in T^c are also isomorphic, and the i-th elements of vectors associated with R and R'' are identical. If R'' is not removed by the subtree merge procedure, it exactly is R'. We then certainly have $\mathbf{p}_{j+1}[i] = \mathbf{p}_{j+1}'[i]$, and the subtrees rooted at N_{j+1} and N_{j+1}' in T^c are certainly isomorphic. Otherwise, the subtree rooted at R'' is removed and finally replaced by the subtree rooted at R'. Thus, there must exists a sequence of nodes $R_1, R_2, \cdots, R_m$ in T, where $R_1 = R''$ and $R_m = R'$, such that for $1 \leq k < m$, R_k and R_{k+1} are on the rightmost paths of branches B_i^N and $B_i^{N'}$, respectively, with N' being a child of N, and the subtrees rooted at R_k and R_{k+1} in T^c are isomorphic. By transitivity of $\cong$, the subtrees rooted at R' and R'' in T^c are isomorphic and are thereby isomorphic to the subtree rooted at R in T^c. Moreover, according to Theorem G.1, the i-th elements of vectors associated with each pair of R_k' and R_{k+1}' are identical. By transitivity, we have the i-th elements of vectors associated with R' and R'' are the same, and are further equal to the i-th element of the vector associated with the R. Overall, we have that the subtrees rooted at N_{j+1} and N_{j+1}' in T^c are isomorphic and $\mathbf{p}_{j+1}[i] = \mathbf{p}_{j+1}'[i]$.

(2) Otherwise, we have $\mathbf{p}_j[i] \geq \mathbf{p}[i]$, and consequently, $(N_j, i+1)$ and $(N_j', i+1)$ will be returned at line 17. Thus, $N_{j+1} = N_j$, $N_{j+1}' = N_j'$ and $i_{j+1} = i_{j+1}' = i+1$. The isomorphism between subtrees rooted at N_{j+1} and N_{j+1}' in T^c is ensured by the induction hypothesis. Besides, we can easily derive that $\mathbf{p}_{j+1}[i+1] = \mathbf{p}_{j+1}'[i+1] = 0$. If $i_{j+1} < l$, the **while** loop (lines 8–9) in Procedure Search continues executing and call Forward. We have $\min\{n, n'\} > j+1$. Otherwise, the terminal condition of the **while** loop is reached. Search returns with $n = n' = j+1$. Overall, we have the induction step holds.

Conclusion: By the principle of induction, Lemma M.6 holds for each j. The proof is complete. □

By leveraging the above Lemma M.6, we present the proof for Theorem 7.12 as follows.

Proof. **(Theorem 7.12)** By Lemma M.6, we have the subtree rooted at N_n in T^c and the subtree rooted at $N'_{n'}$ in T^c are isomorphic, which certainly implies $N_n \cong N_{n'}$, i.e., $N \cong N^*$. Let $P = R_1, R_2, \cdots, R_n$ be the nodes on the leftmost path starting from N^* in T', where $R_1 = N^*$, and let $P' = R'_1, R'_2, \cdots, R_n$ be the nodes on the leftmost path starting from N' in T, where $R'_1 = N'$. For ease of notation, we use $\mathbf{p}_j$ and T_j be the vector associated with R_j and the subtree rooted at R_j in T^c, respectively, where $1 \leq j \leq n$, and $\mathbf{p}'_k$ and T'_k be the vector associated with R'_k and the subtree rooted at R'_k in T^c, respectively, where $1 \leq k \leq n'$. We next prove by induction that for $j = 1, 2, \cdots$, it holds (1) either $\min\{n, n'\} > j$ or $n = n' = j$; (2) $\mathbf{p}_j[l-1] = \mathbf{p}'_j[l-1]$; and (3) $T_j \cong T'_j$.

Base case: When $j = 1$, $R_j = N^*$ and $R'_j = N'$. As N' is the copy of N^*, it certainly holds that $\mathbf{p}_j[l-1] = \mathbf{p}'_j[l-1]$ and $T_j \cong T'_j$. Suppose R_j and R'_j represent the gCore Q. If $Q = \emptyset$, R_j and R'_j are leaf nodes of T and T', respectively. Therefore, we have $n = n' = j$. Otherwise, there is at least one node (leaf) in the P and P' after R_j and R'_j, respectively. We then have $\min\{n, n'\} > j$. Overall, the base case holds.

Induction step: Let $j^* \in \mathbb{Z}$ be given and suppose when $j = j^*$, it holds that $\min\{n, n'\} > j$, $\mathbf{p}_j[l-1] = \mathbf{p}'_j[l-1]$ and $T_j \cong T'_j$. R_{j+1} is the child of R_j whose vector $\mathbf{p}_{j+1}$ differs from $\mathbf{p}_j$ in the $(l-1)$-th element. R'_{j+1} is the child of R'_j whose vector $\mathbf{p}'_{j+1}$ differs from $\mathbf{p}'_j$ in the $(l-1)$-th element. According to Lemma M.7, we have $\mathbf{p}_{j+1}[l-1] = \mathbf{p}'_{j+1}[l-1]$ and $T_{j+1} \cong T'_{j+1}$. Certainly, $R_{j+1} \cong R'_{j+1}$. Let Q be the gCore represented by them. Similar to the base case, if $Q = \emptyset$, R_{j+1} and R'_{j+1} are leaf nodes of T and T', respectively, and we have $n = n' = j+1$. Otherwise, there is at least one node (leaf) in the P and P' after R_{j+1} and R'_{j+1}, respectively. Thus, $\min\{n, n'\} > j+1$. Overall, the inductive step holds.

Conclusion: By the principle of induction, we have $n = n'$ and for $j = 1, 2, \cdots, n$; (2) $\mathbf{p}_j[l-1] = \mathbf{p}'_j[l-1]$; and (3) $T_j \cong T'_j$.

$T_j \cong T'_j$ for $1 \leq j \leq n$ implies $R_j \cong R'_j$ for $1 \leq j \leq n$. Thus, for any pair of adjacent nodes R'_j and R'_{j+1} in P' representing gCore Q and Q', respectively, the vertex set $Q - Q'$ associated with the edge between R'_j and R'_{j+1} are retained on the edge between R_j and R_{j+1}. That is, the sets on the leftmost path from N' to a leaf node in T are all retained on the leftmost path from N^* to a leaf in T'. Overall, the proof is completed. □

M.3.11 Proof for Theorem G.1.

Proof. **(Theorem G.1)** Let T^c be the original P-tree. Let T_j and T'_k denote the subtrees rooted R_j and R'_k in T^c, respectively.

We first prove the **sufficiency**. By definition of branch (Definition 7.11), we can easily have that $sd(\mathbf{p}_j) = sd(\mathbf{p}'_k) = i$ (defined in Section M.3.9). As N' is a child of N, $\mathbf{p}_0 \leq \mathbf{p}'_0$. According to Observation 2, $\mathbf{p}_j$ and $\mathbf{p}_0$ share a prefix of length $i-1$, and $\mathbf{p}'_k$ and $\mathbf{p}'_0$ share a prefix of length $i-1$. We thus have $\mathbf{p}_j[i'] \leq \mathbf{p}'_k[i']$ for $1 \leq i' < i$. Together with Condition (2), we have $\mathbf{p}_j[i'] \leq \mathbf{p}'_k[i'] \leq \hat{\mathbf{p}}_j[i']$ for $i' = 1, 2, \cdots, i-1$. Besides, $\mathbf{p}_j[i] = \mathbf{p}'_k[i]$ (Condition (1)). According to Lemma M.4, $T_j \cong T'_k$. The sufficiency thereby holds.

We next prove the **necessity**. For short, we denote B^N_i and $B^{N'}_i$ by B_N and $B_{N'}$, respectively, and denote their corresponding branch in T^c by B^c_N and $B^c_{N'}$, respectively. Let P^c_N and $P^c_{N'}$ be the rightmost path in B^c_N and $B^c_{N'}$, respectively.

If $T_j \cong T'_k$, we prove Condition (2) is true, which serves as the foundation of the following proof for Condition (1). By Definition 7.8, for each node R' in T'_k, there is a node R in T_j such that $R' \cong R$. According to Definition 7.9, we can derive that $\hat{\mathbf{p}}'_k = \hat{\mathbf{p}}_j$, where $\hat{\mathbf{p}}'_k$ is the signature of T'_k. As each node R' in T'_k is a descendants of R'_k, it holds that $\mathbf{p}'_k \leq \mathbf{p}' \leq \hat{\mathbf{p}}'$, where $\hat{\mathbf{p}}'$ is the maximal vector for $[R']$. Thus, we have $\mathbf{p}'_k \leq \hat{\mathbf{p}}'_k$, and further have $\mathbf{p}'_k \leq \hat{\mathbf{p}}_j$. It certainly implies that $\mathbf{p}'_k[i'] \leq \hat{\mathbf{p}}_j[i']$ for $i' = 1, 2, \cdots, i-1$.

We then prove Condition (1) by contradiction. Suppose $\mathbf{p}_j[i] \neq \mathbf{p}'_k[i]$, it holds either $\mathbf{p}_j[i] < \mathbf{p}'_k[i]$ or $\mathbf{p}_j[i] > \mathbf{p}'_k[i]$. When the first case is true, there must exist a node R' in the subsequence of $P^c_{N'}$ before R'_k such that $\mathbf{p}_j[i] = \mathbf{p}'[i]$, where $\mathbf{p}'$ is the vector associated with R'. Obviously, we have $\mathbf{p}'[i'] = \mathbf{p}'_k[i'] \leq \hat{\mathbf{p}}_j[i']$ for $1 \leq i' < i$. According to the sufficiency we have proved above, the subtree rooted at R_j is isomorphic to the subtree rooted at R'. By transitivity of $\cong$, the subtrees rooted at R' and R'_k are isomorphic, which contradicts with the fact that R'_k is a descendant of R'. When the second case is true, there must exist a node R' with vector $\mathbf{p}'$ in the subsequence of $P^c_{N'}$ after R'_k such that $\mathbf{p}_j[i] = \mathbf{p}'[i]$. Similar to the first case, the subtree rooted at R' is isomorphic to the subtree rooted at R_j, and is further isomorphic to the subtree rooted at R'_k. It is clearly also a contradiction. Overall, we have $\mathbf{p}_j[i] = \mathbf{p}'_k[i]$, and thereby the necessity holds. The proof for Theorem G.1 is complete. □

M.3.12 Proof for Theorem G.2.

Proof. **(Theorem G.2)** For ease of explanation, let us first introduce some notations and facts:

(1) Let T be the P+-tree and T^c be its original P-tree. For short, we denote B^N_i and $B^{N'}_i$ by B_N and $B_{N'}$, respectively, and denote their corresponding branch in T^c by B^c_N and $B^c_{N'}$, respectively. Let P_N and $P_{N'}$ be the rightmost path in B_N and $B_{N'}$, respectively, and let P^c_N and $P^c_{N'}$ be the rightmost path in B^c_N and $B^c_{N'}$, respectively. As described in Section 7.3, subtree elimination procedure may remove some vertices from P^c_N and $P^c_{N'}$, making P_N and $P_{N'}$ being a subsequence of P^c_N and $P^c_{N'}$, respectively. We use S_j to denote the subsequence in B^c_N between R_j and R_{j+1}, and use S'_k to denote the subsequence in $B^c_{N'}$ between R_k and R_{k+1}. All nodes in S_j and S'_k are eliminated, as the premises of which, the subtrees rooted at all nodes in S_j (in the original P-tree T) are isomorphic, which are isomorphic to the subtree rooted at R_{j+1} in T, and the subtrees rooted at all nodes in S'_k (in the original P-tree T) are isomorphic, which are isomorphic to the subtree rooted at R'_{k+1} in T.

(2) In the following, we will use notations R, R' and R'' be denote nodes in P^c_N or $P^c_{N'}$ in the rest of the proof. Unless otherwise stated, their vectors are denoted by $\mathbf{p}$, $\mathbf{p}'$ and $\mathbf{p}''$, respectively.

(3) Vectors of all nodes in both P^c_N and $P^c_{N'}$ are monotonic with regard to the relationship $<$ between vectors. Specifically, by Observation 2, the vectors of all nodes in P^c_N share a prefix of length $i-1$, and the vectors of all nodes in $P^c_{N'}$ share a prefix of length $i-1$. However, the i-th elements of vectors of nodes in P^c_N and $P^c_{N'}$ are monotonically increasing, holding

that $\mathbf{p}'[i] = \mathbf{p}[i] + 1$ for any pair of adjacent nodes R and R' in P^c_N or $P^c_{N'}$.

Let T_j and T'_k denote the subtrees rooted R_j and R'_k in T^c, respectively. Based on Condition (2), we have $\mathbf{p}'_k[i'] = \mathbf{p}'_{k-1}[i'] \leq \hat{\mathbf{p}}_j[i']$ for $1 \leq i' < i$. Thus, to prove $T_j \cong T'_k$, according to Theorem G.1, we only need to prove $\mathbf{p}_j[i] = \mathbf{p}'_k[i]$.

We next prove $\mathbf{p}_j[i] = \mathbf{p}'_k[i]$ by contradiction. Suppose $\mathbf{p}_j[i] \neq \mathbf{p}'_k[i]$, it holds either $\mathbf{p}_j[i] < \mathbf{p}'_k[i]$ or $\mathbf{p}_j[i] > \mathbf{p}'_k[i]$. When the first case is true, by Condition (1), there is a node R' in S'_{k-1} such that $\mathbf{p}'[i] = \mathbf{p}_j[i]$. Based on the Condition (2), we have $\mathbf{p}'[i'] = \mathbf{p}'_{k-1}[i'] \leq \hat{\mathbf{p}}_j[i']$ for $i' < i$. Thus, Theorem G.1 ensures that T_j is isomorphic to the subtree rooted at R' in T^c. Let R and R'' be the child of R_j and R' in P^c_N and $P^c_{N'}$, respectively. It clearly holds that $\mathbf{p}[i] = \mathbf{p}''[i]$. As R is in the subtree rooted at R_j, $\hat{\mathbf{p}}_j \leq \hat{\mathbf{p}}$ holds, where $\hat{\mathbf{p}}$ is the signature of the subtree rooted at R in T^c. We can then derive that $\mathbf{p}''[i'] = \mathbf{p}'_{k-1}[i'] \leq \hat{\mathbf{p}}[i']$ based on Condition (2). By Theorem G.1, the subtrees rooted at R and R' in T^c are isomorphic. Since $R \in S'_{k-1}$, R'' must be also in S'_{k-1} or exactly is R'_k. The preceding subtree rooted at R'' in T^c is thereby isomorphic to the preceding subtree rooted at R' in T^c. Notice that the isomorphism between subtrees rooted at R and R'', and R_j and R'' in T^c also imply the isomorphism between preceding subtrees rooted at them in T^c. By transitivity of $\cong$, we have the preceding subtrees rooted at R_j and R in T^c are isomorphic. That is, R_j will be removed by the subtree elimination procedure and no longer exists in P_N, which clearly is a contradiction.

When the second case is true, we have $\mathbf{p}_j[i] > \mathbf{p}'_k[i]$. According to Condition (1), there is a node R in S_{j-1} such that $\mathbf{p}[i] = \mathbf{p}'_k[i]$. Let S be the subsequence of S_{j-1} staring from R. Obviously, the preceding subtrees of all nodes in S (in the original P-tree T^c) are isomorphic and are isomorphic to the preceding subtree rooted at R_j in T^c. We use T_R to denote the subtree rooted at R in T^c. T_j and all preceding subtrees of nodes in S in T constitute T_R. For any node R' in T_R but not in T_j, it must be in the preceding subtree of some node in S, and there exists a node R'' in T_j such that $R' \cong R''$ by Definition 7.8. Based on Definition 7.9, we can derive that $\hat{\mathbf{p}} = \hat{\mathbf{p}}_j$, where $\hat{\mathbf{p}}$ is the signature of T_R in T^c. With Condition (2), we have $\mathbf{p}'_k[i'] = \mathbf{p}'_{k-1}[i'] \leq \hat{\mathbf{p}}[i']$ for $i' < i$. Hence, the subtree rooted at R in T^c and T'_k are isomorphic by Theorem G.1. Let R'' and R' be the child of R and R'_k in P^c_N and $P^c_{N'}$, respectively. Clearly, the i-th elements of $\mathbf{p}''$ and $\mathbf{p}'$ are identical. Since R'' is a child of R, it holds that $\hat{\mathbf{p}}'' \leq \hat{\mathbf{p}}$, where $\hat{\mathbf{p}}''$ is the signature of the subtree rooted at R'' in T^c. Theorem G.1 then ensures that the subtrees rooted at R and R'' in T^c are isomorphic. Since $R \in S_{j-1}$, R'' must be either also in S_{j-1} or exactly R_j. Thus, the preceding subtrees rooted at R and R'' in T^c are isomorphic. Notice that the isomorphism between subtrees rooted at R and R_k, and R'' and R' in T^c also imply the isomorphism between preceding subtrees rooted at them in T^c. By transitivity of $\cong$, the preceding subtrees rooted at R'_k and R' in T^c are isomorphic. That is, R'_k will be eliminated, which is certainly another contradiction.

Overall, we have $\mathbf{p}_j[i] = \mathbf{p}'_k[i]$. The proof for Theorem G.2 is complete. □

M.4 Correctness of Subtree Transplant

It's simple but tedious to give the full proof of the correctness of using Algorithm 3 to solve the GCS problem on a P+-tree or P+-DAG obtained after subtree transplant. Therefore, in the following, we present the proof sketch only.

M.4.1 Subtree transplant for subtree elimination. Let T^c and T be any P-tree and the P+-tree obtained by applying the subtree elimination procedure introduced in Section 7.3. Theorem 7.5 ensures the correctness of applying Algorithm 3 to solve the GCS problem on the T.

Let T' be the P+-tree obtained by applying the same subtree elimination procedure as obtaining T but realized by subtree transplant (introduced in Section 7.4). We can prove by induction on the height of T that $T \cong T'$, which ensures that the vertex sets on the leftmost path of any node R in T are all retained on the leftmost path of its corresponding node $f(R)$ in T', where f is a bijection from nodes of T to nodes of T' defined in Equation 2. Thus, to prove the correctness of using subtree transplant to realize subtree elimination, we only need to show for the query to the gCore corresponding to any node N in T, Procedure Search in Algorithm 3 returns $f(N)$ in T'.

Suppose Procedure Search in Algorithm 3 returns N^* when running on T'. Let $S = N_1, N_2, \cdots, N_l$ be the sequence of nodes returned at line 17 during the execution of Search to locating N in T, and $S' = N'_1, N'_2, \cdots, N'_l$ be the sequence of nodes returned at line 17 during the execution of Search to locating N^* in T'. Obviously, $N_l = N$ and $N'_l = N^*$. We can prove $f(N_i) = N'_i$ for $i = 1, 2, \cdots, l$ by using a similar induction to the one used in the proof for Lemma M.2. Thus, we obtain that $f(N_l) = N'_l$, i.e., $f(N) = N^*$. Overall, Algorithm 3 can be applied to solve the GCS problem correctly on the P+-tree obtained after subtree transplant.

M.4.2 Subtree transplant for subtree merge. Let T^c be any P-tree and T^+ be the P+-tree obtained by repeatedly applying the subtree elimination procedure to T^c until no more nodes can be removed. After applying the subtree merge procedure to T^+, we obtain a P+-DAG, called T. Theorem 7.12 ensures the correctness of applying Algorithm 3 to solve the GCS problem on the T.

Let T' be the P+-DAG obtained by applying the same subtree emerge procedure to T^+ as obtaining T but realized by subtree transplant. $T \cong T'$ can be proved by induction on the height of T. Thus, the vertex sets on the leftmost path of any node R in T are all retained on the leftmost path of $f(R)$ in T', where f is a bijection from nodes of T to nodes of T' defined in Equation 2. We next show that for the query to the gCore corresponding to any node N in T, Procedure Search in Algorithm 3 returns $f(N)$ in T'.

Assume that Procedure Search in Algorithm 3 returns N^* when running on T'. Let $O = (N_1, i_1), (N_2, i_2), \cdots, (N_n, i_n)$ be the sequence of outputs of Procedure Forward at either line 15 or line 17 during the execution of Search to locating N in T, and $O' = (N'_1, i'_1), (N'_2, i'_2), \cdots, (N'_{n'}, i'_{n'})$ be the sequence of outputs of Procedure Forward at either line 15 or line 17 during the execution of Search to locating N^* in T'. We have $N_{n'} = N$ and $N'_{n'} = N^*$. It's easy to prove $n = n'$, $i_j = i'_j$ and $f(N_j) = N'_j$ for $1 \leq j \leq n$ by induction on j. Therefore, we have $f(N) = N^*$. By collecting all nodes in the vertex sets on the leftmost path of N^*, we certainly obtain the target core corresponding to N. Overall, Algorithm 3 can be applied to solve the GCS problem correctly on the P+-DAG obtained after subtree transplant.

N COMPLEXITY ANALYSIS

N.1 Time and Space Complexity of Algorithm 1

N.1.1 Time Complexity. We will show that Algorithm 1 (GCS) runs in time $O(|\mathcal{M}|+l|V_l|)$, where $|\mathcal{M}| = \sum_{i=1}^{l}|V_i|+|E(\mathcal{G})|+|E(C)|$. In the following, we consider its efficient implementation presented in Algorithm 4 in Appendix B.

Let us first analyze the time overhead of the two subroutines Remove (lines 29-34) and Peel (lines 35-44). It's obvious that Remove runs in constant $O(1)$ time. In Peel, line 36 contributes $O(1)$ to the algorithm. We add superscripts *old* and *new* to s_i and e_i to distinguish their values at the beginning and end of the execution of Peel, respectively. Clearly, the **while** loop in lines 37–44 repeats $e_i^{new} - s_i^{old}$ times. Therefore, line 38 and line 44 cost $O(e_i^{new} - s_i^{old})$ time. Within the **while** loop, there is a **for** loop in lines 39–43 repeating $O(|N_i(v)|)$ times for $v \in vert_i[s_i^{old}, e_i^{new})$, with each using $O(1)$ time to run the conditional statements in lines 40–43. Overall, the **while** loop (lines 37–44), also Peel, costs $O(\sum_{s_i^{old}\le j<e_i^{new}}|N_i(vert_i[j])|)$ time.

We next consider the main body of Algorithm 4. In the initialization phase (lines 1–9), line 2 and line 6 run in $O(1)$ time, and hence contributes $O(l)$ to the algorithm. The **for** loops in lines 3–5 and lines 7–9 both have size $|V_i|$ with constant time $O(1)$ bodies. Therefore, the two loops cost $O(\sum_{i=1}^{l}|V_i|)$ time. The initialization phase runs in $O(\sum_{i=1}^{l}|V_i|)$ time.

In the vertex peeling phase (lines 10–27), suppose that the **while** loop in lines 10–27 repeats T times. As each run removes at least one vertex from V_l, we have $T \le |V_l|$. Therefore, line 19 and line 27 contribute $O(|V_l|)$ and $O(l|V_l|)$ time to the algorithm, respectively. Assume that after calling Peel in line 11 in the $(t-1)$-th and the t-th runs of the **while** loop, the value of e_l becomes e_l^{t-1} and e_l^t, respectively. We certainly have that the value of s_l equals e_l^{t-1} before calling Peel in the line 11 in the t-th run because of line 19. Therefore, line 11 takes $O(\sum_{e_l^{t-1}\le j<e_l^t}|N_l(vert_l[j])|)$ time in the t-th run, and totally uses time:

$$O\left(\sum_{t=1}^{T}\sum_{j=e_l^{t-1}}^{e_l^t}|N_l(vert_l[j])|\right)=O\left(\sum_{j=e_l^0}^{e_l^T}|N_l(vert_l[j])|\right)=O\left(\sum_{v\in V_l}|N_l(v)|\right)=O(|E_l|).$$

The conditional statements in lines 15–18 run in constant $O(1)$ time. It is executed for each cross layer edge from layer G_l to each G_i at most once, and hence the **for** loop in lines 12–18 contributes time $O(\sum_{i=1}^{l-1}|E_{l,i}|)$ in all repetitions of the **while** loop (lines 10–27). Similar to line 11, Peel in line 21 costs $O(\sum_{i=1}^{l-1}|E_i|)$ time across all runs of the **for** loop in lines 20–26. The conditional statements in lines 23–26 run in constant $O(1)$ time, and are executed for each cross layer edge from each layer G_i to G_l at most once. Therefore, the **for** loop in lines 22–26 contributes time $O(\sum_{i=1}^{l-1}|E_{i,l}|)$ in total. Overall, the vertex peeling phase takes $O((l+1)|V_l| + \sum_{i=1}^{l}|E_i| + \sum_{i=1}^{l-1}|E_{l,i}| + \sum_{i=1}^{l-1}|E_{i,l}|)$ time.

To sum up, we have the time complexity of Algorithm 4 (Algorithm 1) is:

$$O\left(\sum_{i=1}^{l}|V_i|+(l+1)|V_l|+\sum_{i=1}^{l}|E_i|+\sum_{i=1}^{l-1}|E_{l,i}|+\sum_{i=1}^{l-1}|E_{i,l}|\right)$$
$$=O\left(\sum_{G_i\in\mathcal{G}}|V_i|+|E(\mathcal{G})|+|E(C|+l|V_l|\right)=O\left(|\mathcal{M}|+l|V_l|\right)$$

N.1.2 Space Complexity. We will show that Algorithm 1 runs in space $O(l\cdot V_{max})$, where $V_{max} = \max_{1\le i\le l}|V_i|$. In the following, we also consider its implementation in Algorithm 4.

Algorithm 4 sets up two arrays $vert_i$ and pos_i and two offsets s_i and e_i for each layer $G_i \in \mathcal{G}$, which takes $O(\sum_{i=1}^{l}|V_i|)$ space. Maintaining the intra-layer degree for each vertex in $\mathcal{M}$ requires $Q(\sum_{i=1}^{l}|V_i|)$ space. Maintaining each cross-layer degree $d_i(v)$ for $v \in V_l$ takes $Q(\sum_{i=1}^{l-1}|V_l|)$ space. Maintaining the cross-layer degree $d_l(v)$ for $v \in V_i, 1 \le i < l$ takes $Q(\sum_{i=1}^{l-1}|V_i|)$ space. Overall, we have the space complexity of Algorithm 4 (Algorithm 1) is:

$$O\left(\sum_{i=1}^{l}|V_i|+\sum_{i=1}^{l-1}|V_l|+\sum_{i=1}^{l-1}|V_i|\right)=O\left(l\cdot V_{max}\right)$$

N.2 Time and Space Complexity of Algorithm 2

Let us first introduce the foundation of our analysis. Given P-tree T nested in the $\mathbf{k}$-node in the KP-tree and let R be the root of T, we conceptually divide T into a series of leftmost paths by the following rules:

R1 If $l = 2$, T itself is R's leftmost path and is therefore no longer divided.

R2 Otherwise, suppose that R has n children, namely $R_1, R_2, \cdots, R_n$, and among which R_1 is the leftmost child of R. T is then divided into the leftmost path of R and a series of subtrees rooted at $R_2, \cdots, R_n$, respectively, say $T_2, \cdots, T_n$. Thereafter, each T_i with $2 \le i \le n$ is divided recursively using the same way.

Recall that Algorithm 2 generates the nodes of a P-tree in depth-first order, which ensures that each node is generated followed by the generation of its leftmost child. That is, for any node N in a P-tree T, all cores corresponding to nodes on the leftmost path of N are generated consecutively. We next consider the computation on each leftmost path as a unit. Lemma N.1 presents the computational overhead for Algorithm 2 to process a leftmost path. And Lemma N.2 gives the total number of leftmost paths that constitute T.

Lemma N.1. *Given P-tree T nested in the $\mathbf{k}$-node in the KP-tree, let N be any node in T, Algorithm 2 computes all cores on the leftmost path of N in $O(|\mathcal{M}| + l|V_l| + l|F_{l-1}|)$ time and $O(|l\cdot V_{max}|)$ space.*

Proof. Let $N_1, N_2, \cdots, N_m$ be the leftmost path starting from N in T, in which $N_1 = N$ and N_m is a leaf node. It's obvious that m is bounded by $O(|F_{l-1}|)$. For $1 \le j \le m$, we use $\mathbf{p}_j$ and Q_j to denote the vector associated with N_i and the $(\mathbf{k}, \mathbf{p}_j)$-core corresponding to N_j, respectively. It's easy to see that $Q_1, Q_2, \cdots, Q_m$ are computed by a series of consecutively recursive calls of procedure PTreeDFS (lines 4–13) in Algorithm 2.

Time complexity. We first analyze the time overhead. Recall that GCS in line 5 is implemented by Algorithm 4 introduced in Appendix B based on a series of arrays. Specifically, for each layer G_i, we have set up arrays $vert_i$ and pos_i to facilitate vertex peeling.

When the GCS terminates, $vert_i[e_i, |V_i|)$ consists all vertex remaining in layer G_i. It enables Algorithm 2 to share these arrays among all calls to GCS to compute $Q_1, Q_2, \cdots, Q_m$. For ease of notation, we use e_i^{old} to denote the value of e_i when the execution of GCS terminates. When Q_j, where $1 < j \leq m$, is about to be computed in line 5, we simply set both s_i and e_i to the value of e_i^{old} obtained in the last call of GCS to compute Q_{j-1} for $1 \leq i \leq l$. After that, vertex peeling (lines 10–27) in Algorithm 4 can be continually performed by using the existing arrays $vert_i$ and pos_i. When the call of GCS to compute Q_m is finished, e_i becomes $|V_i|$. It's easy to see that during the execution of line 5 across all recursive calls to PTreeDFS to compute $Q_1, Q_2, \cdots Q_m$, lines 40–43 in Algorithm 4 are executed twice for each intra-layer edge in $\mathcal{M}$, lines 15–18 are executed once for each cross-layer edge from G_l to each G_i, and lines 23–26 are executed once for each cross-layer edge from each G_i to G_l. Therefore, these three parts contribute $|E(\mathcal{G})| + |E(C)|$ time to the algorithm in total. As the **while** loop in lines 10–27 repeats at most $|V_l|$ times, line 19 and line 27 cost $|V_l|$ and $l|V_l|$ time, respectively. Overall, the vertex peeling phase (lines 10–27) in the execution of GCS (line 5) needs $O((l+1) \cdot |V_l| + |E(\mathcal{G})| + |E(C|)$ time. Lines 1–6 of the initialization phase (lines 1–9) of GCS is only performed at most once (for computing Q_1), which requires $O(\sum_{i=1}^{l} |V_i|)$ time as stated in Appendix B. The conditional statement in lines 8–9 of GCS are executed at most once for each vertex in $\mathcal{M}$, and therefore cost $O(\sum_{i=1}^{l} |V_i|)$ time. Together with $O(l \cdot m)$ time to assign s_i and e_i each time before performing vertex peeling, the initialization phase needs $O(\sum_{i=1}^{l} |V_i| + l \cdot m)$ time in total. In addition, each execution of line 5 also corresponds to a call to Procedure ToFrac in lines 22-23, which can be implemented in $O(l)$ time by keeping a reverse index of each F_i. To sum up, we have that the overall time overhead of line 5 is:

$$O\left((l+1) \cdot |V_l| + |E(\mathcal{G})| + |E(C| + \sum_{i=1}^{l} |V_i| + l \cdot m + l \cdot m\right) = O(|\mathcal{M}| + l|V_l| + l|F_{l-1}|)$$

Besides line 5, lines 10–11 are executed for each computed Q_i, and therefore contribute $O(l \cdot m)$ time. Finally, we have that Algorithm 2 computes all cores on the leftmost path of N in $O(|\mathcal{M}| + l|V_l| + l|F_{l-1}| + l \cdot m) = O(|\mathcal{M}| + l|V_l| + l|F_{l-1}|)$ time.

Space complexity. We next analyze the space overhead. Line 5 takes $O(l \cdot V_{max})$ space as all arrays set up for facilitating vertex peeling are shared among calls to GCS to compute $Q_1, Q_2, \cdots, Q_m$. Lines 10–11 use $O(l)$ space to keep the newly generated vector $\mathbf{p}'$. Therefore, we have the space complexity for Algorithm 2 to compute all cores on the leftmost path of N is $O(l \cdot V_{max})$.

Overall, the lemma holds. □

Lemma N.2. *Any P-tree T will be divided into $O(\prod_{i=1}^{l-2} |F_i|)$ leftmost paths by applying rule R1 or R2.*

Proof. When $l = 2$, by rule R1, T consists of only one leftmost path, i.e., itself. Therefore, the lemma holds.

We next consider the case when $l > 2$. For any node N in T, let $\mathbf{p}$ be the vector associated with N. It's obvious that the leftmost path starting from N contributes to T if and only if $end0(\mathbf{p}) \geq 1$. Therefore, the number of leftmost paths constituting T is equal to the number of $\mathbf{p}$-nodes in T such that $end0(\mathbf{p}) \geq 1$, and is further equal to the number of vectors $\mathbf{v} \in F_1 \times F_2 \times \cdots \times F_{l-2}$, which is $O(\prod_{j=1}^{l-2} |F_j|)$. The lemma thus holds. □

N.2.1 Time Complexity of Algorithm 2. By Lemma N.1 and Lemma N.2, it takes $O(\prod_{i=1}^{l-2} |F_i| \cdot (|\mathcal{M}| + l|V_l| + l|F_{l-1}|))$ time to process the P-tree nested in each $\mathbf{k}$-node in the KP-tree. Thus, line 15 runs in $O(\prod_{i=1}^{l} \kappa(G_i) \cdot \prod_{i=1}^{l-2} |F_i| \cdot (|\mathcal{M}| + l|V_l| + l|F_{l-1}|))$ time. Lines 19–20 are executed once for each $\mathbf{k}$-node in the KP-tree, and therefore consume $O(l \cdot \prod_{i=1}^{l} \kappa(G_i))$ time. Notice that the pseudocode for generating each F_i is omitted in Algorithm 2, however it takes $O(l \cdot |V_l| + \sum_{i=1}^{l-1} d_i^2 \log d_i)$ time which will be analyzed in Section N.4.1. Overall, we have the time complexity of Algorithm 2 is:

$$O\left(\prod_{i=1}^{l} \kappa(G_i) \cdot \prod_{i=1}^{l-2} |F_i| \cdot (|\mathcal{M}| + l|V_l| + l|F_{l-1}|) + l \cdot \prod_{i=1}^{l} \kappa(G_i) + l \cdot |V_l| + \sum_{i=1}^{l-1} d_i^2 \log d_i\right)$$
$$= O\left(\prod_{i=1}^{l} \kappa(G_i) \cdot \prod_{i=1}^{l-2} |F_i| \cdot (|\mathcal{M}| + l|V_l|) + l \cdot \prod_{i=1}^{l} \kappa(G_i) \cdot \prod_{i=1}^{l-1} |F_i| + \sum_{i=1}^{l-1} d_i^2 \log d_i\right)$$

N.2.2 Space Complexity of Algorithm 2. As the space for GCS (line 5) to process each leftmost path, also each P-tree, can be reused, line 15 takes $O(l \cdot V_{max})$ space according to Lemma N.1. Lines 19–20 use $O(l)$ space to keep the newly generated vector $\mathbf{k}'$. In addition, we also need $O(l \cdot \sum_{i=1}^{l-1} |F_i|)$ space to store each F_i and its reverse index. Overall, we have that the space complexity of Algorithm 2 is:

$$O\left(l \cdot V_{max} + 1 + l \cdot \sum_{i=1}^{l-1} |F_i|\right) = O\left(l \cdot \left(V_{max} + \sum_{i=1}^{l-1} |F_i|\right)\right)$$

N.3 Time and Space complexity of Algorithm 3

Time Complexity. We will show that Algorithm 3 runs in $O(\sum_{i=1}^{l-1} |F_i| + |Q|)$ time, where $|Q|$ is the size of the output $(\mathbf{k}, \mathbf{p})$-core.

Line 1 costs $O(1)$ time to perform a hash table lookup. Procedure Search (line 3) visits each node on the path from the root node of the P-tree to the $\hat{\mathbf{p}}$-node in $O(h_1)$ time, where h_1 is the length of the path. Procedure Recover (line 4) visits each edge on the leftmost path starting from the $\hat{\mathbf{p}}$-node to a dummy leaf node and collects vertex sets on each edge using $O(h_2 + |Q|)$, where h_2 is the length of the leftmost path of the $\hat{\mathbf{p}}$-node. Obviously, $h_1 + h_2$ is bounded by the height of the P-tree $O(\sum_{i=1}^{l-1} |F_i|)$. We then have the time complexity of Algorithm 3 is $O(\sum_{i=1}^{l-1} |F_i| + |Q|)$.

Space Complexity. Obviously, except for the space for storing the KP-tree, Procedure Search runs in $O(1)$ space, and Procedure Recover runs in $O(|Q|)$ space. We then have the space complexity of Algorithm 3 is $O(|Q|)$.

N.4 Time and Space Complexity of Algorithm 5

N.4.1 Time Complexity. We will show the time complexity of Algorithm 5 is $O(\prod_{i=1}^{l} \kappa(G_i) \cdot \prod_{i=1}^{l-1} |F_i| \cdot (|\mathcal{M}| + l|V_l|) + \sum_{i=1}^{l-1} d_i^2 \log d_i)$.

Let us first analyze the time overhead of the subroutine GenFrac (lines 11–16) in Algorithm 5. Line 12 runs in $O(1)$ time to initialize F_i. In our realization, we first identify distinct values of possible cross-layer degrees of vertices in V_l to avoid generating duplicate fractions. It can be finished in $O(|V_l|)$ time. After that, the **for** loop in lines 13–15 repeats at most d_i times, where d_i is the maximum cross-layer degree. For each distinct cross-layer degree d, the **for** loop in lines 14–15 insert $d + 1$ fractions into set F_i. Therefore, $O(d_i^2)$ insertions are performed in all repetitions of **for** loop in lines 13–15, which requires $O(d_i^2 \log d_i)$ time. Overall, GenFrac runs in $O(|V_l| + d_i^2 \log d_i)$ time.

We next consider the main body of Algorithm 5. Line 1 runs in time $O(1)$ to initialize R. Line 2 generates all possible values of $\mathbf{k}$ with each costing $O(l)$ time, and therefore contributes $O(l \cdot \prod_{i=1}^{l} \kappa(G_i))$ time. The **for** loop in lines 3–4 repeats for each layer G_i, where $1 \leq i < l$, and hence takes $O(\sum_{i=1}^{l-1}(|V_l| + d_i^2 \log d_i))$. Line 5 generates all possible values of $\mathbf{p}$ with each costing $O(l-1)$ time, and altogether contributes $O(l \cdot \prod_{i=1}^{l-1} |F_i|)$ time. For each combination of $\mathbf{k} \in K$ and $\mathbf{p} \in P$, line 8 is executed once in $O(|\mathcal{M}| + l|V_l|)$ time. To sum up, line 8 contributes $O(\prod_{i=1}^{l} \kappa(G_i) \cdot \prod_{i=1}^{l-1} |F_i| \cdot (|\mathcal{M}| + l|V_l|)$ time. Overall, we have the time complexity of Algorithm 5 is:

$$
\begin{aligned}
&O\left(1 + l \cdot \prod_{i=1}^{l} \kappa(G_i) + \sum_{i=1}^{l-1} \left(|V_l| + d_i^2 \log d_i\right) + l \cdot \prod_{i=1}^{l-1} |F_i| + \prod_{i=1}^{l} \kappa(G_i) \cdot \prod_{i=1}^{l-1} |F_i| \cdot (|\mathcal{M}| + l|V_l|)\right) \\
=&O\left(\prod_{i=1}^{l} \kappa(G_i) \cdot \prod_{i=1}^{l-1} |F_i| \cdot (|\mathcal{M}| + l|V_l|) + \sum_{i=1}^{l-1} d_i^2 \log d_i\right)
\end{aligned}
$$

N.4.2 Space Complexity. We will show that the space complexity of Algorithm 5 is $O(l \cdot (\prod_{i=1}^{l} \kappa(G_i) + \prod_{i=1}^{l-1} |F_i| + V_{max}))$, where $V_{max} = \max_{1 \leq i \leq l} |V_i|$.

Let us first analyze the space overhead of the subroutine GenFrac. As stated in Section N.4.1, we detect distinct values of cross-layer degrees of vertices in V_l, which can be realized in $O(|V_l|)$ space. Maintaining all generated fractions in F_i costs $O((l-1) \cdot |F_i|)$ space. Overall, GenFrac takes $O(|V_l| + l|F_i|)$ space.

We next consider the main body of Algorithm 5. As it has to keep all generated vectors $\mathbf{k} \in K$ and $\mathbf{p} \in P$, $O(l \cdot (\prod_{i=1}^{l} \kappa(G_i) + \prod_{i=1}^{l-1} |F_i|))$ space is needed. Besides, it consumes $O(|V_l| + \max_{1 \leq i < l} l|F_i|)$ space to run GenFrac (line 4) as the space for each run can be reused. Similarly, all runs of GCS in line 8 cost $O(l \cdot V_{max})$ space. To sum up, we have the space complexity of Algorithm 5 is:

$$
\begin{aligned}
&O\left(l \cdot \left(\prod_{i=1}^{l} \kappa(G_i) + \prod_{i=1}^{l-1} |F_i|\right) + |V_l| + \max_{1 \leq i < l} l \cdot |F_i| + l \cdot V_{max}\right) = \\
=&O\left(l \cdot \left(\prod_{i=1}^{l} \kappa(G_i) + \prod_{i=1}^{l-1} |F_i| + V_{max}\right)\right)
\end{aligned}
$$

N.5 Time Complexity of Subtree Merge

We will show that the time complexity to apply subtree merge to compact a P+-tree is $O\left(\sum_{i=2}^{l-1} \prod_{j=1}^{i} |F_j|\right)$.

Given P+-tree T nested in the $\mathbf{k}$-node in the KP-tree, for any branch B_i^N in T, the subtree merge procedure introduced in Section 7.5 detects strong isomorphic subtrees within B_i^N and $B_i^{N'}$, where N' is the parent of N. As we have analyzed in Section G.2, it can be realized by the pivot-based optimization in $O(m+n)$ time, where m and n are lengths of the rightmost path of N and N' in B_i^N and $B_i^{N'}$, respectively. It's obvious that both m and n are bounded by $O(|F_i|)$. If strong isomorphic subtrees are identified, the merge operation, which is implemented by the subtree transplant approach introduced in Section 7.4, can be applied at most once to B_i^N, and therefore costs $O(1)$ time. Thus, it takes $O(|F_i|)$ time for subtree merge to process branch B_i^N.

We conceptually divide branches in T to be examined by subtree merge into the following sets: $R_2, R_3, \cdots, R_{l-1}$, in which R_i consists of all branches B_i^N of $\mathbf{p}$-nodes N such that $end0(\mathbf{p}) \geq l - i$. As analyzed in the proof of Lemma N.2, the number of $\mathbf{p}$-nodes in the P-tree satisfying $end0(\mathbf{p}) \geq l - i$ is equal to the number of vectors $\mathbf{v} \in F_1 \times F_2 \times \cdots \times F_{i-1}$, which is $\prod_{j=1}^{i-1} |F_j|$. Therefore, we have $|R_i| = \prod_{j=1}^{i-1} |F_j|$. Since each branch in R_i contributes $|F_i|$ time to the whole subtree merge process, we have the time complexity of applying subtree merge is:

$$
O\left(\sum_{i=2}^{l-1}\left(\prod_{j=1}^{i-1} |F_j| \cdot |F_i|\right)\right) = O\left(\sum_{i=2}^{l-1} \prod_{j=1}^{i} |F_j|\right)
$$