2018 IEEE 34th International Conference on Data Engineering

Diversified Coherent Core Search on Multi-Layer Graphs

Rong Zhu, Zhaonian Zou, and Jianzhong Li
Harbin Institute of Technology, Harbin, Heilongjiang 150001, China
{rzhu, znzou, lijzh}@hit.edu.cn

***Abstract*—Mining dense subgraphs on multi-layer graphs is an interesting problem, which has witnessed lots of applications in practice. To overcome the limitations of the quasi-clique-based approach, we propose *d*-coherent core (*d*-CC), a new notion of dense subgraph on multi-layer graphs, which has several elegant properties. We formalize the diversified coherent core search (DCCS) problem, which finds *k d*-CCs that can cover the largest number of vertices. We propose a greedy algorithm with an approximation ratio of $1 - 1/e$ and two search algorithms with an approximation ratio of 1/4. The experiments verify that the search algorithms are faster than the greedy algorithm and produce comparably good results as the greedy algorithm in practice. As opposed to the quasi-clique-based approach, our DCCS algorithms can fast detect larger dense subgraphs that cover most of the quasi-clique-based results.**

I. Introduction

Dense subgraph mining, that is, finding vertices cohesively connected by internal edges, is an important task in graph mining. In the literature, many dense subgraph notions have been formalized [9], e.g. clique, quasi-clique, k-core, k-truss, k-plex, and k-club. Meanwhile, a large number of dense subgraph mining algorithms have also been proposed.

In many real-world scenarios, a graph often contains various types of edges, which represent various types of relationships between entities. For example, in biological networks, interactions between genes can be detected by different methods [7]; in social networks, users can interact through different social media [12]. In [4][11], such a graph with multiple types of edges is modelled as a *multi-layer graph*, where each layer independently accommodates a certain type of edges.

Finding dense subgraphs on multi-layer graphs has witnessed many real-world applications.

Application 1 (Biological Module Discovery). In biological networks, densely connected vertices (genes or proteins), also known as biological modules, play an important role in detecting protein complexes and co-expression clusters [7]. Due to data noise, there often exist a number of spurious biological interactions (edges), so a group of vertices only cohesively connected by interactions detected by a certain method may not be a convincing biological module. To filter out the effects of spurious interactions and make the detected modules more reliable, biologists detect interactions using multiple methods, i.e., build a multi-layer biological network, where each layer contains interactions detected by a certain method. A set of vertices is regarded as a reliable biological module if they are simultaneously densely connected on multiple layers [7].

Application 2 (Story Identification in Social Media.) Social media such as Twitter and Facebook is updating with numerous new posts every day. A story in a social media is an event capturing popular attention recently [1]. Stories can be identified by leveraging some real-world entities involved in them, such as people, locations, companies, and products. To identify them, scientists often abstract all new posts at each moment as a snapshot graph, where each vertex represents an entity, and each edge links two entities if they frequently occur together in these new posts. A number of snapshot graphs are maintained in a time window. After that, a story can be identified by finding a group of strongly connected entities on multiple snapshot graphs [1]. Obviously, this is an instance of finding dense subgraphs on multi-layer graphs.

Different from dense subgraph mining on single-layer graphs, dense subgraphs on multi-layer graphs must be evaluated by the following two metrics: 1) ***Density***: The interconnections between the vertices must be sufficiently dense on some individual layers. 2) ***Support***: The vertices must be densely connected on a sufficiently large number of layers.

In the literature, the most representative and widely used notion of dense subgraphs on multi-layer graphs is *cross-graph quasi-clique* [4][11][19]. On a single-layer graph, a vertex set Q is a *γ-quasi-clique* if every vertex in Q is adjacent to at least $\gamma(|Q|-1)$ vertices in Q, where $\gamma \in [0, 1]$. Given a set of graphs $G_1, G_2, \ldots, G_n$ with the same vertices (i.e., layers in our terminology), $\gamma \in [0, 1]$ and $min_s \in \mathbb{N}$, a vertex set Q is a *cross-graph quasi-clique* if Q is a γ-quasi-clique on all of $G_1, G_2, \ldots, G_n$ and $|Q| \geq min_s$. Although the cross-graph quasi-clique notion considers both density and support, it has several intrinsic limitations inherited from γ-quasi-cliques.

The lower bound of a vertex's degree in a γ-quasi-clique Q is $\gamma(|Q|-1)$, which linearly increases with $|Q|$. This constraint is too strict for large dense subgraphs in real graphs. The diameter of a γ-quasi-clique is often too small. As proved in [11], the diameter of a γ-quasi-clique is at most 2 for $\gamma \geq 0.5$. Hence, in cross-graph quasi-clique mining, a large dense subgraph tends to be decomposed into many quasi-cliques. It leads to the following limitations:

1) Finding all quasi-cliques is computationally hard and is not scalable to large graphs [4].

2) Quasi-cliques are useful in the study of micro-clusters (e.g. motifs [6]) but are not suitable for studying large clusters (e.g. communities). To alleviate this problem, quasi-cliques are merged together to restore large dense subgraphs in post-

2375-026X/18/$31.00 ©2018 IEEE
DOI 10.1109/ICDE.2018.00069

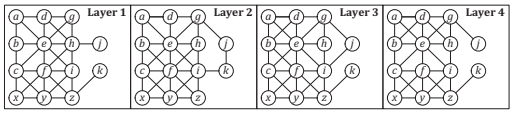


Fig. 1. Example of 4-Layer Graph.

processing [9]. However, the merging process not only takes additional time but also the quality of the restored subgraphs depends on the discovered quasi-cliques. Since γ and min_s indirectly affects the properties of the restored subgraphs, it is difficult for users to specify appropriate parameters. For example, in the 4-layer graph in Fig. 1, the vertex set $Q = \{a, b, c, d, e, f, g, h, i, x, y, z\}$ naturally induces a large dense subgraph on all layers. However, for $\gamma \geq 0.5$ and $min_s = 6$, the restored subgraph is $\{c, f, i, x, y, z\}$, which miss many vertices in Q.

Hence, there naturally arises the first question:

Q1: *What is a better notion of dense subgraphs on multi-layer graphs, which can avoid the limitations of cross-graph quasi-cliques?*

Additionally, as discovered in [4], dense subgraphs on multi-layer graphs have significant overlaps. For practical usage, it is better to output a small subset of *diversified* dense subgraphs with little overlaps. Ref. [4] proposes an algorithm to find diversified cross-graph quasi-cliques. One of our goal in this paper is to find dense subgraphs on even larger multi-layer graphs. There will be even more dense subgraphs, so the problem of finding diversified dense subgraphs will be even more critical. Hence, we face the second question:

Q2: *How to design efficient algorithms to find diversified dense subgraphs according to the new notion?*

To deal with the first question **Q1**, we present a new notion called d-*coherent core* (d-*CC* for short) to characterize dense subgraphs on multi-layer graphs. It is extended from the d-core notion on single-layer graphs [3]. Specifically, given a multi-layer graph $\mathcal{G}$, a subset L of layers of $\mathcal{G}$ and $d \in \mathbb{N}$, the d-CC with respect to (w.r.t. for short) L is the maximum vertex subset S such that each vertex in S is adjacent to at least d vertices in S on all layers in L. The d-CC w.r.t. L is unique. The d-CC notion is a natural fusion of density and support. In comparison with cross-graph quasi-clique, the constraint d on the degree of the vertices in a d-CC is independent of the size of the d-CC. There is no limit on the diameter of a d-CC, and a d-CC often consists of a large number of densely connected vertices. The d-CC notion has the following advantages:

1) A d-CC can be computed in linear time w.r.t. the size of a multi-layer graph.

2) A d-CC itself is a large dense subgraph. It is unnecessary to use a post-processing phase to restore large dense subgraphs. The parameter d directly controls the properties of the expected results. For example, in Fig. 1, for $d = 3$, the d-CC on all layers is $\{a, b, c, d, e, f, g, h, i, x, y, z\}$, which is directly the large dense subgraph in the multi-layer graph.

3) The d-CC notion inherits the hierarchy property of d-core: The $(d+1)$-CC w.r.t. L is a subset of the d-CC w.r.t. L; The d-CC w.r.t. L is a subset of the d-CC w.r.t. L' if $L' \subseteq L$.

The d-CC notion overcomes the limitations of cross-graph quasi-cliques. Based on this notion, we formalize the *diversified coherent core search (DCCS)* problem that finds dense subgraphs on multi-layer graphs with little overlaps: Given a multi-layer graph $\mathcal{G}$, a minimum degree threshold d, a minimum support threshold s, and the number k of d-CCs to be detected, the DCCS problem finds k most diversified d-CCs recurring on at least s layers of $\mathcal{G}$. Like [2][4], we assess the diversity of the k discovered d-CCs by the number of vertices they cover and try to maximize the diversity of these d-CCs. We prove that the DCCS problem is NP-complete.

To deal with the second question **Q2**, we propose a series of approximation algorithms for the DCCS problem. First, we propose a simple greedy algorithm, which finds k d-CCs in a greedy manner. The algorithm has an approximation ratio of $1 - 1/e$. However, it must compute all candidate d-CCs and therefore is not scalable to large multi-layer graphs.

To prune unpromising candidate d-CCs early, we propose two search algorithms, namely the bottom-up search algorithm and the top-down search algorithm. In both algorithms, the process of generating candidate d-CCs and the process of updating diversified d-CCs interact with each other. Many d-CCs that are unpromising to appear in the final results are pruned in early stage. The bottom-up and top-down algorithms adopt different search strategies. In practice, the bottom-up algorithm is preferable if $s < l/2$, and the top-down algorithm is preferable if $s \geq l/2$, where l is the number of layers. Both of the algorithms have an approximation ratio of $1/4$.

We conducted extensive experiments on a variety of datasets to evaluate the proposed algorithms and obtain the following results: 1) The bottom-up and top-down algorithms are 1–2 orders of magnitude faster than the greedy algorithm for small and large s, respectively. 2) The practical approximation quality of the bottom-up and top-down algorithms is comparable to that of the greedy algorithm. 3) Our DCCS algorithms outperform the cross-graph quasi-clique mining algorithms [4][11][19] on multi-layer graphs in terms of both execution time and result quality.

II. Problem Definition

Multi-Layer Graphs. A *multi-layer graph* is a set of graphs $\{G_1, G_2, \ldots, G_l\}$, where l is the number of layers, and G_i is the graph on layer i. Without loss of generality, we assume that $G_1, G_2, \ldots, G_l$ contain the same set of vertices because if a vertex is missing from layer i, we can add it to G_i as an isolated vertex. Hence, a multi-layer graph $\{G_1, G_2, \ldots, G_l\}$ can be equivalently represented by $(V, E_1, E_2, \ldots, E_l)$, where V is the universal vertex set, and E_i is the edge set of G_i.

Let $V(G)$ and $E(G)$ be the vertex set and the edge set of graph G, respectively. For a vertex $v \in V(G)$, let $N_G(v) = \{u | (v, u) \in E(G)\}$ be the set of *neighbors* of v in G, and let $d_G(v) = |N_G(v)|$ be the *degree* of v in G. The subgraph of G induced by a vertex subset $S \subseteq V(G)$ is $G[S] = (S, E[S])$, where $E[S]$ is the set of edges with both endpoints in S.

Given a multi-layer graph $\mathcal{G} = (V, E_1, E_2, \ldots, E_l)$, let $l(\mathcal{G})$ be the number of layers of $\mathcal{G}$, $V(\mathcal{G})$ the vertex set of $\mathcal{G}$, and $E_i(\mathcal{G})$ the edge set of the graph on layer i. The multi-layer

subgraph of $\mathcal{G}$ induced by a vertex subset $S \subseteq V(\mathcal{G})$ is $\mathcal{G}[S] = (S, E_1[S], E_2[S], \ldots, E_l[S])$, where $E_i[S]$ is the set of edges in E_i with both endpoints in S.

***d*-Coherent Cores.** We define the notion of *d-coherent core* (*d-CC*) on a multi-layer graph by extending the *d-core* notion on a single-layer graph [3]. A graph G is *d-dense* if $d_G(v) \geq d$ for all $v \in V(G)$, where $d \in \mathbb{N}$. The *d-core* of graph G, denoted by $C^d(G)$, is the maximum subset $S \subseteq V(G)$ such that $G[S]$ is d-dense. As stated in [3], $C^d(G)$ is unique, and $C^d(G) \subseteq C^{d-1}(G) \subseteq \cdots \subseteq C^1(G) \subseteq C^0(G)$ for $d \in \mathbb{N}$.

For ease of notation, let $[n] = \{1, 2, \ldots, n\}$, where $n \in \mathbb{N}$. Let $\mathcal{G}$ be a multi-layer graph and $L \subseteq [l(\mathcal{G})]$ be a non-empty subset of layer numbers. For $S \subseteq V(\mathcal{G})$, the induced subgraph $\mathcal{G}[S]$ is d-dense w.r.t. L if $G_i[S]$ is d-dense for all $i \in L$. The *d-coherent core* (*d-CC*) of $\mathcal{G}$ w.r.t. L, denoted by $C^d_L(\mathcal{G})$, is the maximum subset $S \subseteq V(\mathcal{G})$ such that $\mathcal{G}[S]$ is d-dense w.r.t. L. Similar to d-core, the concept of d-CC has the following properties.

Property 1 (Uniqueness): Given a multi-layer graph $\mathcal{G}$ and a subset $L \subseteq [l(\mathcal{G})]$, $C^d_L(\mathcal{G})$ is unique for $d \in \mathbb{N}$.

Property 2 (Hierarchy): Given a multi-layer graph $\mathcal{G}$ and a subset $L \subseteq [l(\mathcal{G})]$, we have $C^d_L(\mathcal{G}) \subseteq C^{d-1}_L(\mathcal{G}) \subseteq \cdots \subseteq C^1_L(\mathcal{G}) \subseteq C^0_L(\mathcal{G})$ for $d \in \mathbb{N}$.

Property 3 (Containment): Given a multi-layer graph $\mathcal{G}$ and two subsets $L, L' \subseteq [l(\mathcal{G})]$, if $L \subseteq L'$, we have $C^d_{L'}(\mathcal{G}) \subseteq C^d_L(\mathcal{G})$ for $d \in \mathbb{N}$.

Due to space limitations, we put the proofs of all properties, lemmas and theorems in Appendix A of the full paper [21].

Problem Statement. Given a multi-layer graph $\mathcal{G}$, a minimum degree threshold $d \in \mathbb{N}$ and a minimum support threshold $s \in \mathbb{N}$, let $\mathcal{F}_{d,s}(\mathcal{G})$ be the set of d-CCs of $\mathcal{G}$ w.r.t. all subsets $L \subseteq [l(\mathcal{G})]$ such that $|L| = s$. When $\mathcal{G}$ is large, $|\mathcal{F}_{d,s}(\mathcal{G})|$ is often very large, and a large number of d-CCs in $\mathcal{F}_{d,s}(\mathcal{G})$ significantly overlap with each other. For practical usage, it is better to output k diversified d-CCs with little overlaps, where k is a number specified by users. Like [2][4], we assess the diversity of the discovered d-CCs by the number of vertices they cover and try to maximize the diversity of these d-CCs. Let the *cover set* of a collection of sets $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$ be $\mathsf{Cov}(\mathcal{R}) = \bigcup_{i=1}^{n} R_i$. We formally define the *Diversified Coherent Core Search (DCCS) problem* as follows.

Given a multi-layer graph $\mathcal{G}$, a minimum degree threshold d, a minimum support threshold s, and the number k of d-CCs to be discovered, find the subset $\mathcal{R} \subseteq \mathcal{F}_{d,s}(\mathcal{G})$ such that 1) $|\mathcal{R}| = k$, and 2) $|\mathsf{Cov}(\mathcal{R})|$ is maximized. The d-CCs in $\mathcal{R}$ are called the *top-k diversified d-CCs* of $\mathcal{G}$ on s layers.

Theorem 1: The DCCS problem is NP-complete.

III. The Greedy Algorithm

This section proposes a simple greedy algorithm with an approximation ratio of $1 - 1/e$. Before describing the algorithm, we present the following lemma based on Property 3. The lemma enables us to remove irrelevant vertices earlier.

Lemma 1 (Intersection Bound): Given a multi-layer graph $\mathcal{G}$ and two subsets $L_1, L_2 \subseteq [l(\mathcal{G})]$, we have $C^d_{L_1 \cup L_2}(\mathcal{G}) \subseteq C^d_{L_1}(\mathcal{G}) \cap C^d_{L_2}(\mathcal{G})$ for $d \in \mathbb{N}$.

Algorithm GD-DCCS($\mathcal{G}, d, s, k$)
1: $\mathcal{G} \leftarrow$ VertexDeletion($\mathcal{G}, d, s$)
2: $\mathcal{F} \leftarrow \emptyset$; $\mathcal{R} \leftarrow \emptyset$
3: **for** $i \leftarrow 1$ to $l(\mathcal{G})$ **do**
4: compute $C^d(G_i)$ on G_i
5: **for** each $L \subseteq [l(\mathcal{G})]$ such that $|L| = s$ **do**
6: $S \leftarrow \bigcap_{i \in L} C^d(G_i)$
7: $C^d_L(\mathcal{G}) \leftarrow$ dCC($\mathcal{G}[S], L, d$)
8: $\mathcal{F} \leftarrow \mathcal{F} \cup \{C^d_L(\mathcal{G})\}$
9: **for** $j \leftarrow 1$ to k **do**
10: $C^* \leftarrow \arg\max_{C \in \mathcal{F}}(|\mathsf{Cov}(\mathcal{R} \cup \{C\})| - |\mathsf{Cov}(\mathcal{R})|)$
11: $\mathcal{R} \leftarrow \mathcal{R} \cup \{C^*\}$; $\mathcal{F} \leftarrow \mathcal{F} - \{C^*\}$
12: **return** $\mathcal{R}$

Fig. 2. The GD-DCCS Algorithm.

The greedy algorithm GD-DCCS is described in Fig. 2. The input is a multi-layer graph $\mathcal{G}$ and $d, s, k \in \mathbb{N}$. GD-DCCS works as follows. Line 1 preprocesses $\mathcal{G}$ by Procedure VertexDeletion, which will be described later. Line 2 initializes the candidate d-CC set $\mathcal{F}$ and the result set $\mathcal{R}$ to be empty. Lines 3–4 compute the d-core $C^d(G_i)$ on each layer G_i by the algorithm in [3]. Indeed, we have $C^d_{\{i\}}(\mathcal{G}) = C^d(G_i)$.

In order to find $C^d_L(\mathcal{G})$ for each $L \subseteq [l(\mathcal{G})]$ with $|L| = s$, we first compute the intersection $S = \bigcap_{i \in L} C^d(G_i)$ (line 6). By Lemma 1, we have $C^d_L(\mathcal{G}) \subseteq S$. Thus, we compute $C^d_L(\mathcal{G})$ by Procedure dCC on the induced subgraph $\mathcal{G}[S]$ instead of on $\mathcal{G}$ (line 7) and add $C^d_L(\mathcal{G})$ to $\mathcal{F}$ (line 8). Procedure dCC is similar to the procedure of computing the d-core on a single-layer graph [3]. Due to space limitations, we describe the details of dCC in Appendix B of [21].

Lines 9–11 select k d-CCs from $\mathcal{F}$ in a greedy manner. Each time, we select the d-CC $C^* \in \mathcal{F}$ that maximizes $|\mathsf{Cov}(\mathcal{R} \cup \{C^*\})| - |\mathsf{Cov}(\mathcal{R})|$, add C^* to $\mathcal{R}$ (line 10) and remove C^* from $\mathcal{F}$ (line 11). Finally, $\mathcal{R}$ is output as the result (line 12).

Let $l = l(\mathcal{G})$, $n = |V(\mathcal{G})|$ and $m = |\bigcup_{i=1}^{l} E_i(\mathcal{G})|$. Procedure dCC in line 7 runs in $O(ns + ms)$ time as shown in Appendix B of [21]. Line 10 runs in $O(n|\mathcal{F}|)$ time since computing $|\mathsf{Cov}(\mathcal{R} \cup \{C\})| - |\mathsf{Cov}(\mathcal{R})|$ takes $O(n)$ time for each $C \in \mathcal{F}$. In addition, $|\mathcal{F}| = \binom{l}{s}$. Therefore, the time complexity of GD-DCCS is $O((ns + ms + kn)\binom{l}{s})$, and the space complexity is $O(n\binom{l}{s})$.

Theorem 2: The approximation ratio of GD-DCCS is $1 - \frac{1}{e}$.

Preprocessing. In line 1 of GD-DCCS, we apply Procedure VertexDeletion to remove unpromising vertices from $\mathcal{G}$. Let $\mathsf{Num}(v)$ denote the number of layers i such that $v \in C^d(G_i)$. If $\mathsf{Num}(v) < s$, v cannot be contained in any d-CCs $C^d_L(\mathcal{G})$ with $|L| = s$. Thus, we can iteratively remove all such vertices from $\mathcal{G}$ until $\mathsf{Num}(v) \geq s$ for all vertices v remaining in $\mathcal{G}$. Due to space limitations, we describe Procedure VertexDeletion in Appendix C of [21].

Limitations. As verified by the experimental results in Section VI, GD-DCCS is not scalable to large multi-layer graphs. This is due to the following reasons: 1) GD-DCCS must keep all candidate d-CCs in $\mathcal{F}$. As $l(\mathcal{G})$ increases, $|\mathcal{F}|$ grows exponentially. When $\mathcal{F}$ cannot fit in main memory, the algorithm incurs large amounts of I/Os. 2) The exponential growth in $|\mathcal{F}|$ significantly increases the time of selecting k diversified d-CCs from $\mathcal{F}$ (lines 9–11). 3) The candidate d-CC generation phase (lines 2–8) is separate from the diversified d-CC selection phase (lines 9–11). There is no guidance on candidate generation, so a large number of unpromising candidates are generated in vain.

IV. The Bottom-Up Algorithm

This section proposes a bottom-up approach to the DCCS problem. In this approach, the candidate d-CC generation phase and the top-k diversified d-CC selection phase are *interleaved*. On one hand, we maintain a set of temporary top-k diversified d-CCs and use each newly generated d-CC to update them. On the other hand, we guide candidate d-CC generation by the temporary top-k diversified d-CCs.

In addition, candidate d-CCs are generated in a bottom-up manner. Like the frequent pattern mining algorithm [18], we organize all d-CCs by a search tree and search candidate d-CCs on it. The bottom-up d-CC generation has the following advantage: If the d-CC w.r.t. subset L ($|L| < s$) is unlikely to improve the quality of the temporary top-k diversified d-CCs, the d-CCs w.r.t. all L' such that $L \subseteq L'$ and $|L'| = s$ need not be generated. As verified by the experimental results in Section VI, the bottom-up approach reduces the search space by 80%–90% in comparison with the greedy algorithm and thus saves large amount of time. Moreover, the bottom-up DCCS algorithm has an approximation ratio of $1/4$.

A. Maintenance of Top-k Diversified d-CCs

Let $\mathcal{R}$ be a set of temporary top-k diversified d-CCs. Initially, $\mathcal{R} = \emptyset$. To improve the quality of $\mathcal{R}$, we try to update $\mathcal{R}$ whenever we find a new candidate d-CC C. In particular, we update $\mathcal{R}$ with C by one of the following rules:

Rule 1: If $|\mathcal{R}| < k$, C is added to $\mathcal{R}$.

Rule 2: For $C' \in \mathcal{R}$, let $\Delta(\mathcal{R}, C') = C' - \mathsf{Cov}(\mathcal{R} - \{C'\})$, that is, $\Delta(\mathcal{R}, C')$ is vertex set in $\mathsf{Cov}(\mathcal{R})$ exclusively covered by C'. Let $C^*(\mathcal{R}) = \arg\min_{C' \in \mathcal{R}} |\Delta(\mathcal{R}, C')|$, that is, $C^*(\mathcal{R})$ exclusively covers the least number of vertices among all d-CCs in $\mathcal{R}$. We replace $C^*(\mathcal{R})$ with C if $|\mathcal{R}| = k$ and

$$|\mathsf{Cov}((\mathcal{R} - \{C^*(\mathcal{R})\}) \cup \{C\})| \geq (1 + \tfrac{1}{k})|\mathsf{Cov}(\mathcal{R})|. \quad (1)$$

On input $\mathcal{R}$ and C, Procedure Update tries to update $\mathcal{R}$ with C using the rules above. The details of Update is described in Appendix D of the full paper [21]. By using two index structures, Update runs in $O(\max\{|C|, |C^*(\mathcal{R})|\})$ time.

B. Bottom-Up Candidate Generation

Candidate d-CCs $C^d_L(\mathcal{G})$ with $|L| = s$ are generated in a bottom-up fashion. As shown in Fig. 4, all d-CCs $C^d_L(\mathcal{G})$ are conceptually organized by a search tree, in which $C^d_L(\mathcal{G})$ is the parent of $C^d_{L'}(\mathcal{G})$ if $L \subset L'$, $|L'| = |L| + 1$, and the only number $\ell \in L' - L$ satisfies $\ell > \max(L)$, where $\max(L)$ is the largest number in L (specially, $\max(\emptyset) = -\infty$). Conceptually, the root of the search tree is $C^d_\emptyset(\mathcal{G}) = V(\mathcal{G})$.

The d-CCs in the search tree are generated in a depth-first order. First, we find the d-core $C^d(G_i)$ on each single layer G_i. By definition, we have $C^d_{\{i\}}(\mathcal{G}) = C^d(G_i)$. Then, starting from $C^d_{\{i\}}(\mathcal{G})$, we generate the descendants of $C^d_{\{i\}}(\mathcal{G})$. The depth-first search is realized by recursive Procedure BU-Gen in Fig. 3. In general, given a d-CC $C^d_L(\mathcal{G})$ as input, we first expand L by adding a layer number j such that $\max(L) < j \leq l(\mathcal{G})$. Let $L' = L \cup \{j\}$. By Lemma 1, we have $C^d_{L'}(\mathcal{G}) \subseteq C^d_L(\mathcal{G}) \cap C^d_{\{j\}}(\mathcal{G}) = C^d_L(\mathcal{G}) \cap C^d(G_j)$. Thus, we compute $C^d_{L'}(\mathcal{G})$ on the induced subgraph $\mathcal{G}[C^d_L(\mathcal{G}) \cap C^d(G_j)]$ by Procedure dCC described in Section III. Next, we process $C^d_{L'}(\mathcal{G})$ according to the following cases:

Procedure BU-Gen($\mathcal{G}, d, s, k, L, C^d_L(\mathcal{G}), L_Q, \mathcal{R}$)
1: $L_P \leftarrow \{j | \max(L) < j \leq l(\mathcal{G})\} - L_Q$; $L_R \leftarrow \emptyset$
2: **if** $|\mathcal{R}| < k$ **then**
3: **for** $j \in L_P$ **do**
4: $L' \leftarrow L \cup \{j\}$
5: $C^d_{L'}(\mathcal{G}) \leftarrow \mathsf{dCC}(\mathcal{G}[C^d_L(\mathcal{G}) \cap C^d(G_j)], L', d)$
6: **if** $|L'| = s$ **then**
7: Update($\mathcal{R}, C^d_{L'}(\mathcal{G})$)
8: **else**
9: $L_R \leftarrow L_R \cup \{j\}$
10: **else if** $|\mathcal{R}| = k$ **then**
11: sort $j \in L_P$ in descending order of $|C^d_L(\mathcal{G}) \cap C^d(G_j)|$
12: **for** each j in the sorted L_P **do**
13: **if** $|C^d_L(\mathcal{G}) \cap C^d(G_j)| < \frac{1}{k}|\mathsf{Cov}(\mathcal{R})| + |\Delta(\mathcal{R}, C^*(\mathcal{R}))|$ **then**
14: **break**
15: **else**
16: $L' \leftarrow L \cup \{j\}$
17: $C^d_{L'}(\mathcal{G}) \leftarrow \mathsf{dCC}(\mathcal{G}[C^d_L(\mathcal{G}) \cap C^d(G_j)], L', d)$
18: **if** $|L'| = s$ **then**
19: Update($\mathcal{R}, C^d_{L'}(\mathcal{G})$)
20: **else**
21: **if** $C^d_{L'}(\mathcal{G})$ satisfies Eq. (1) **then**
22: $L_R \leftarrow L_R \cup \{j\}$
23: **if** $|L| < s$ **then**
24: **for** $j \in L_R$ **do**
25: $L' \leftarrow L \cup \{j\}$
26: BU-Gen($\mathcal{G}, d, s, k, L', C^d_{L'}(\mathcal{G}), L_Q \cup (L_P - L_R), \mathcal{R}$)

Fig. 3. The BU-Gen Procedure.

Case 1: If $|L'| = s$, we update $\mathcal{R}$ with $C^d_{L'}(\mathcal{G})$.

Case 2: If $|L'| < s$ and $|\mathcal{R}| < k$, we recursively call BU-Gen to generate the descendants of $C^d_{L'}(\mathcal{G})$.

Case 3: If $|L'| < s$ and $|\mathcal{R}| = k$, we check if $C^d_{L'}(\mathcal{G})$ satisfies Eq. (1) to update $\mathcal{R}$. If not satisfied, none of the descendants of $C^d_{L'}(\mathcal{G})$ is qualified to be a candidate, so we prune the entire subtree rooted at $C^d_{L'}(\mathcal{G})$; otherwise, we recursively call BU-Gen to generate the descendants of $C^d_{L'}(\mathcal{G})$. The correctness is guaranteed by the following lemma.

Lemma 2 (Search Tree Pruning): For a d-CC $C^d_L(\mathcal{G})$, if $C^d_L(\mathcal{G})$ does not satisfy Eq. (1), none of the descendants of $C^d_L(\mathcal{G})$ can satisfy Eq. (1).

Pruning Methods: To further improve the efficiency, if $|\mathcal{R}| = k$, we order the layer numbers $j > \max(L)$ in decreasing order of $|C^d_L(\mathcal{G}) \cap C^d(G_j)|$ and generate $C^d_{L \cup \{j\}}(\mathcal{G})$ according to this order of j. For some j, if $|C^d_L(\mathcal{G}) \cap C^d(G_j)| < \frac{1}{k}|\mathsf{Cov}(\mathcal{R})| + |\Delta(\mathcal{R}, C^*(\mathcal{R}))|$, we can stop searching the subtrees rooted at $C^d_{L \cup \{j\}}(\mathcal{G})$ and $C^d_{L \cup \{j'\}}(\mathcal{G})$ for all j' succeeding j in the order. The correctness is ensured by the following lemma.

Lemma 3 (Order-based Pruning): For a d-CC $C^d_L(\mathcal{G})$ and $j > \max(L)$, if $|C^d_L(\mathcal{G}) \cap C^d(G_j)| < \frac{1}{k}|\mathsf{Cov}(\mathcal{R})| + |\Delta(\mathcal{R}, C^*(\mathcal{R}))|$, $C^d_{L \cup \{j\}}(\mathcal{G})$ cannot satisfy Eq. (1).

Another method is the *layer pruning*. For $\max(L) < j \leq l(\mathcal{G})$, if $|\mathcal{R}| = k$ and $C^d_{L \cup \{j\}}(\mathcal{G})$ does not satisfy Eq. (1), we need not generate $C^d_{L'}(\mathcal{G})$ for all L' such that $L \cup \{j\} \subseteq L' \subseteq [l(\mathcal{G})]$. The correctness is guaranteed by the following lemma.

Lemma 4 (Layer Pruning): For a d-CC $C^d_L(\mathcal{G})$ and $j > \max(L)$, if $C^d_{L \cup \{j\}}(\mathcal{G})$ does not satisfy Eq. (1), $C^d_{L' \cup \{j\}}(\mathcal{G})$ cannot satisfy Eq. (1) for all L' such that $L \subseteq L' \subseteq [l(\mathcal{G})]$.

Fig. 3 describes the pseudocode of Procedure BU-Gen, which naturally follows the steps presented above. Here, we make a few necessary remarks. The input L_Q is the set of layer numbers that cannot be used to expand L. They are obtained according to Lemma 4 during generating the ascendants of $C^d_L(\mathcal{G})$. Thus, the layer numbers possible to be added to L are $L_P = \{j | \max(L) < j \leq l(\mathcal{G})\} - L_Q$ (line 1). In BU-

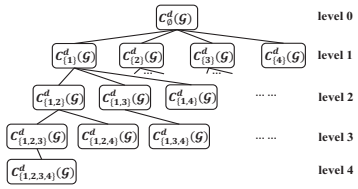


Fig. 4. Bottom-Up Search Tree.

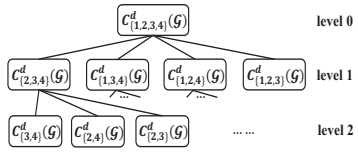


Fig. 5. Top-Down Search Tree.

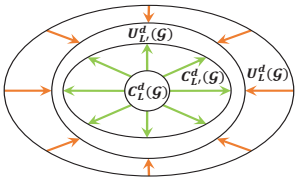


Fig. 6. Relationships between $C^d_L(\mathcal{G})$, $U^d_L(\mathcal{G})$, $C^d_{L'}(\mathcal{G})$ and $U^d_{L'}(\mathcal{G})$.

Algorithm BU-DCCS($\mathcal{G}, d, s, k$)
1: $\mathcal{G} \leftarrow$ VertexDeletion($\mathcal{G}, d, s$)
2: $\mathcal{R} \leftarrow$ InitTopK($\mathcal{G}, d, s, k$)
3: sort all layer numbers in descending order of $|C^d(G_i)|$, where $i \in [l(\mathcal{G})]$
4: BU-Gen($\mathcal{G}, d, s, k, \emptyset, V(\mathcal{G}), \emptyset, \mathcal{R}$)
5: **return** $\mathcal{R}$

Fig. 7. The BU-DCCS Algorithm.

Gen, we use a set L_R to record the layer numbers that are actually added to L (lines 9 and 22). In lines 24–26, for each $j \in L_R$, we make a recursive call to BU-Gen to generate the descendants of $C^d_{L\cup\{j\}}(\mathcal{G})$. By Lemma 4, the layer numbers that cannot be added to L' are $L_Q \cup (L_P - L_R)$.

C. The Bottom-Up Algorithm

Fig. 7 describes the complete bottom-up DCCS algorithm BU-DCCS. Given a multi-layer graph $\mathcal{G}$ and three parameters $d, s, k \in \mathbb{N}$, we can solve the DCCS problem by calling BU-Gen($\mathcal{G}, d, s, k, \emptyset, V(\mathcal{G}), \emptyset, \mathcal{R}$) (line 4). To further speed up the algorithm, the preprocessing method (Procedure VertexDeletion) proposed in Section III is applied in line 1. In addition, we propose two additional preprocessing methods.

Sorting Layers. We sort the layers of $\mathcal{G}$ in descending order of $|C^d(G_i)|$, where $1 \le i \le l(\mathcal{G})$. Intuitively, the larger $|C^d(G_i)|$ is, the more likely G_i contains a large candidate d-CC. Although there is no theoretical guarantee on the effectiveness of this preprocessing method, it is indeed effective in practice. Line 3 of BU-DCCS applies this preprocessing method.

Initialization of $\mathcal{R}$. The pruning techniques in BU-Gen are not applicable unless $|\mathcal{R}| = k$, so a good initial state of $\mathcal{R}$ can greatly enhance the pruning power. We develop a greedy procedure InitTopK to initialize $\mathcal{R}$ so that $|\mathcal{R}| = k$. We describe Procedure InitTopK in Appendix E of [21]. Line 2 of BU-DCCS initializes $\mathcal{R}$ by Procedure InitTopK.

Theorem 3: The approximation ratio of BU-DCCS is $1/4$.

V. The Top-Down Algorithm

The bottom-up algorithm traverses a search tree from the root down to level s. When $s \ge l(\mathcal{G})/2$, the efficiency of the algorithm degrades significantly. As verified by the experiments in Section VI, the performance of the bottom-up algorithm is close to or even worse than the greedy algorithm when $s \ge l(\mathcal{G})/2$. To handle this issue, we propose a top-down approach for the DCCS problem when $s \ge l(\mathcal{G})/2$.

In this section, we assume $s \ge l(\mathcal{G})/2$. In the top-down algorithm, we maintain a temporary top-k result set $\mathcal{R}$ and update it in the same way as in the bottom-up algorithm. However, candidate d-CCs are generated in a top-down manner. The reverse in search direction makes the techniques in the bottom-up algorithm no longer suitable. Therefore, we propose a new candidate d-CC generation method and a series of new pruning techniques suitable for the top-down search. The top-down algorithm has an approximation ratio of $1/4$. As verified by the experiments in Section VI, the top-down algorithm is superior to the other algorithms when $s \ge l(\mathcal{G})/2$.

A. Top-Down Candidate Generation

We first introduce how to generate d-CCs in a top-down manner. In the top-down algorithm, all d-CCs are conceptually organized as a search tree as illustrated in Fig. 5, where $C^d_L(\mathcal{G})$ is the parent of $C^d_{L'}(\mathcal{G})$ if $L' \subset L$, $|L| = |L'|+1$, and the only layer number $\ell \in L - L'$ satisfies $\ell > \max([l(\mathcal{G})] - L)$. Except the root $C^d_{[l(\mathcal{G})]}$, all d-CCs in the search tree has a unique parent. We generate candidate d-CCs by depth-first searching the tree from the root down to level s and update the temporary result set $\mathcal{R}$ during search.

Let $C^d_L(\mathcal{G})$ be the d-CC currently visited in DFS, where $|L| > s$. We must generate the children of $C^d_L(\mathcal{G})$. By Property 3 of d-CCs, we have $C^d_L(\mathcal{G}) \subseteq C^d_{L'}(\mathcal{G})$ for all $L' \subseteq L$. Thus, to generate $C^d_{L'}(\mathcal{G})$, we only have to add some vertices to $C^d_L(\mathcal{G})$ but need not delete any vertex from $C^d_L(\mathcal{G})$.

To this end, we associate $C^d_L(\mathcal{G})$ with a vertex set $U^d_L(\mathcal{G})$. $U^d_L(\mathcal{G})$ must contain the vertices in all descendants $C^d_S(\mathcal{G})$ of $C^d_L(\mathcal{G})$ such that $|S| = s$. $U^d_L(\mathcal{G})$ serves as the scope for searching for the descendants of $C^d_L(\mathcal{G})$. We call $U^d_L(\mathcal{G})$ the *potential vertex set* w.r.t. $C^d_L(\mathcal{G})$. Obviously, we have $C^d_L(\mathcal{G}) \subseteq U^d_L(\mathcal{G})$. Initially, $U^d_{[l(\mathcal{G})]}(\mathcal{G}) = V(\mathcal{G})$. Section V-B will describe how to shrink $U^d_L(\mathcal{G})$ to $U^d_{L'}(\mathcal{G})$ for $L' \subseteq L$, so we have $U^d_{L'}(\mathcal{G}) \subseteq U^d_L(\mathcal{G})$ if $L' \subseteq L$. The relationships between $C^d_L(\mathcal{G})$, $U^d_L(\mathcal{G})$, $C^d_{L'}(\mathcal{G})$ and $U^d_{L'}(\mathcal{G})$ are illustrated in Fig. 6. The arrows in Fig. 6 indicates that $C^d_{L'}(\mathcal{G})$ is expanded from $C^d_L(\mathcal{G})$, and $U^d_{L'}(\mathcal{G})$ is shrunk from $U^d_L(\mathcal{G})$. Keeping this in mind, we focus on top-down candidate generation in this subsection. Sections V-B and V-C will describe how to compute $U^d_{L'}(\mathcal{G})$ and $C^d_{L'}(\mathcal{G})$, respectively.

The top-down candidate d-CC generation is implemented by the recursive procedure TD-Gen in Fig. 8. Let $L_R = \{j | \max([l(\mathcal{G})] - L) < j \le l(\mathcal{G})\} \cap L$ be the set of layer numbers possible to be removed from L (line 1). For each $j \in L_R$, let $L' = L - \{j\}$. We have that $C^d_{L'}(\mathcal{G})$ is a child of $C^d_L(\mathcal{G})$. We first obtain $U^d_{L'}(\mathcal{G})$ and $C^d_{L'}(\mathcal{G})$ by the methods in Section V-B (line 4) and Section V-C (line 5), respectively. Next, we process $C^d_{L'}(\mathcal{G})$ based on the following cases:

Case 1 (lines 9–10): If $|\mathcal{R}| < k$ and $|L'| = s$, we update $\mathcal{R}$ with $C^d_{L'}(\mathcal{G})$ by Rule 1 specified in Section IV-A.

Case 2 (lines 11–12): If $|\mathcal{R}| < k$ and $|L'| > s$, we recursively call TD-Gen to generate the descendants of $C^d_{L'}(\mathcal{G})$.

Case 3 (lines 20–21): If $|\mathcal{R}| = k$ and $|L'| = s$, we update $\mathcal{R}$ with $C^d_{L'}(\mathcal{G})$ by Rule 2 specified in Section IV-A.

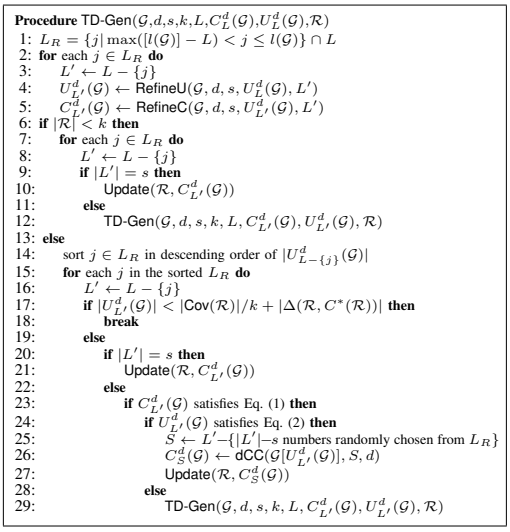


Fig. 8. The TD-Gen Procedure.

Case 4 (lines 22–29): If $|\mathcal{R}| = k$ and $|L'| > s$, we check if $U^d_{L'}(\mathcal{G})$ satisfies Eq. (1) to update $\mathcal{R}$ (line 23). If it is not satisfied, none of the descendants of $C^d_{L'}(\mathcal{G})$ is qualified to be a candidate d-CC, so we prune the entire subtree rooted at $C^d_{L'}(\mathcal{G})$. Otherwise, we recursively call TD-Gen to generate the descendants of $C^d_{L'}(\mathcal{G})$ (line 29). The correctness of the pruning method is guaranteed by the following lemma.

Lemma 5 (Search Tree Pruning): For a d-CC $C^d_L(\mathcal{G})$ and its potential vertex set $U^d_L(\mathcal{G})$, where $|L| > s$, if $U^d_L(\mathcal{G})$ does not satisfy Eq. (1), any descendant $C^d_{L'}(\mathcal{G})$ of $C^d_L(\mathcal{G})$ with $|L'| = s$ cannot satisfy Eq. (1).

Pruning Methods: If $|\mathcal{R}| = k$ (Cases 3 and 4), to further prune the search tree, we order the layer numbers $j \in L_R$ in the descending order of $|U^d_{L-\{j\}}(\mathcal{G})|$ (line 14). For some $j \in L_R$, if $|U^d_{L-\{j\}}(\mathcal{G})| < |\mathsf{Cov}(\mathcal{R})|/k + |\Delta(\mathcal{R}, C^*(\mathcal{R}))|$, we need not to consider all layer numbers in L_R succeeding j and can terminate searching the subtrees rooted at $C^d_{L-\{j\}}(\mathcal{G})$ immediately (lines 17–18). The correctness of this pruning method is ensured by the following lemma.

Lemma 6 (Order-based Pruning): For a d-CC $C^d_L(\mathcal{G})$, its potential vertex set $U^d_L(\mathcal{G})$ and $j > \max([l(\mathcal{G})] - L)$, if $|U^d_{L-\{j\}}(\mathcal{G})| < |\mathsf{Cov}(\mathcal{R})|/k + |\Delta(\mathcal{R}, C^*(\mathcal{R}))|$, any descendant $C^d_{L-\{j\}}(\mathcal{G})$ of $C^d_L(\mathcal{G})$ cannot satisfy Eq. (1).

More interestingly, for Case 4, in some optimistic cases, we need not to search the descendants of $C^d_L(\mathcal{G})$. Instead, we can randomly select a descendant $C^d_S(\mathcal{G})$ of $C^d_L(\mathcal{G})$ with $|S| = s$ to update $\mathcal{R}$ (lines 25–27). The correctness is ensured by the following lemma.

Lemma 7 (Potential Set Pruning): For a d-CC $C^d_L(\mathcal{G})$ and its potential vertex set $U^d_L(\mathcal{G})$, where $|L| > s$, if $C^d_L(\mathcal{G})$ satisfies Eq. (1), and $U^d_L(\mathcal{G})$ satisfies

$$|U^d_L(\mathcal{G})| < (\tfrac{1}{k} + \tfrac{1}{k^2})|\mathsf{Cov}(\mathcal{R})| + (1 + \tfrac{1}{k})|\Delta(\mathcal{R}, C^*(\mathcal{R}))|, \quad (2)$$

the following proposition holds: For any two distinct descendants $C^d_{S_1}(\mathcal{G})$ and $C^d_{S_2}(\mathcal{G})$ of $C^d_L(\mathcal{G})$ such that $|S_1| = |S_2| = s$, if $|\mathcal{R}| = k$ and $\mathcal{R}$ has already been updated by $C^d_{S_1}(\mathcal{G})$, then $C^d_{S_2}(\mathcal{G})$ cannot update $\mathcal{R}$ any more.

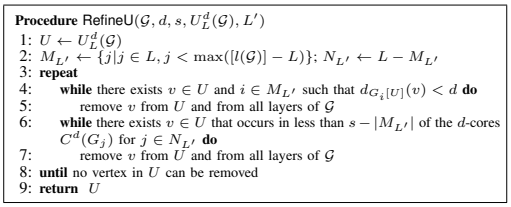


Fig. 9. The RefineU Procedure.

B. Refinement of Potential Vertex Sets

Let $C^d_L(\mathcal{G})$ be the d-CC currently visited by the DFS and $C^d_{L'}(\mathcal{G})$ be a child of $C^d_L(\mathcal{G})$. To generate $C^d_{L'}(\mathcal{G})$, Procedure TD-Gen first refines $U^d_L(\mathcal{G})$ to $U^d_{L'}(\mathcal{G})$ and then generates $C^d_{L'}(\mathcal{G})$ based on $U^d_{L'}(\mathcal{G})$. This subsection introduces how to shrink $U^d_L(\mathcal{G})$ to $U^d_{L'}(\mathcal{G})$.

First we introduce some useful notation. Given a subset of layer numbers $L \subseteq [l(\mathcal{G})]$, we can divide all layer numbers in L into two disjoint classes:

Class 1: By the relationship of d-CCs in the top-down search tree, for any layer number $\ell \in L$ and $\ell < \max([l(\mathcal{G})] - L)$, ℓ will not be removed from L in any descendant of $C^d_L(\mathcal{G})$. Thus, for any descendant $C^d_S(\mathcal{G})$ of $C^d_L(\mathcal{G})$ with $|S| = s$, we have $l \in S$.

Class 2: By the relationship of d-CCs in the top-down search tree, for any layer number $\ell \in L$ and $\ell > \max([l(\mathcal{G})] - L)$, ℓ can be removed from L to obtain a descendant of $C^d_L(\mathcal{G})$. Thus, for a descendant $C^d_S(\mathcal{G})$ of $C^d_L(\mathcal{G})$ with $|S| = s$, it is undetermined whether $\ell \in S$.

Let M_L and N_L denote the Class 1 and Class 2 of layer numbers w.r.t. L, respectively. Procedure RefineU in Fig. 9 refines $U^d_L(\mathcal{G})$ to $U^d_{L'}(\mathcal{G})$. Let $U = U^d_L(\mathcal{G})$ (line 1). First, we obtain $M_{L'}$ and $N_{L'}$ w.r.t. L' (line 2). Then, we repeat the following two refinement methods to remove irrelevant vertices from U until no vertices can be removed any more (lines 3–8). Finally, U is output as $U^d_{L'}(\mathcal{G})$ (line 9).

Refinement Method 1 (lines 4–5): For each layer number $i \in M_{L'}$, we have $i \in S$ for all descendants $C^d_S(\mathcal{G})$ of $C^d_{L'}(\mathcal{G})$ with $|S| = s$. Note that $C^d_S(\mathcal{G})$ must be d-dense in G_i. Thus, if the degree of a vertex v in $G_i[U]$ is less than d, we have $v \notin C^d_S(\mathcal{G})$, so we can remove v from U and $\mathcal{G}$.

Refinement Method 2 (lines 6–7): If a vertex $v \in U$ is contained in a descendant $C^d_S(\mathcal{G})$ of $C^d_L(\mathcal{G})$ with $|S| = s$, v must occur in all the d-cores $C^d(G_i)$ for $i \in M_{L'}$ and must occur in at least $s - |M_{L'}|$ of the d-cores $C^d(G_j)$ for $j \in N_{L'}$. Therefore, if v occurs in less than $s - |M_{L'}|$ of the d-cores $C^d(G_j)$ for $j \in N_{L'}$, we can remove v from U and $\mathcal{G}$.

C. Refinement of d-CCs

Let $C^d_L(\mathcal{G})$ be the d-CC currently visited by the DFS and $C^d_{L'}(\mathcal{G})$ be a child of $C^d_L(\mathcal{G})$, where $|L| > s$. Since $C^d_{L'}(\mathcal{G}) \subseteq U^d_{L'}(\mathcal{G})$, Procedure dCC in Section III can find $C^d_{L'}(\mathcal{G})$ on $\mathcal{G}[U^d_{L'}(\mathcal{G})]$ from scratch. However, this straightforward method is not efficient. In this subsection, we propose an more efficient algorithm to construct $C^d_{L'}(\mathcal{G})$ by adopting two techniques: 1) An index structure that helps eliminate more vertices in $U^d_{L'}(\mathcal{G})$ irrelevant to $C^d_{L'}(\mathcal{G})$. 2) A search strategy with early termination to find $C^d_{L'}(\mathcal{G})$ efficiently.

Index Structure. First, we introduce an index structure that organizes all vertices of $\mathcal{G}$ hierarchically and helps filter out the vertices irrelevant to $C^d_{L'}(\mathcal{G})$ efficiently. Recall that $\mathsf{Num}(v)$ is the number of layers whose d-cores contain v. Values $\mathsf{Num}(v)$ are used to determine the vertices in $U^d_{L'}(\mathcal{G})$ that are not in $C^d_{L'}(\mathcal{G})$. Specifically, for $h \in \mathbb{N}$, let J_h be the set of vertices v iteratively removed from $\mathcal{G}$ due to $\mathsf{Num}(v) \leq h$. Let $I_h = J_h - J_{h-1}$. Obviously, $I_1, I_2, \ldots, I_{l(\mathcal{G})}$ is a disjoint partition of all vertices of $\mathcal{G}$. Based on this partition, we can narrow down the search scope of $C^d_{L'}(\mathcal{G})$ from $U^d_{L'}(\mathcal{G})$ to $U^d_{L'}(\mathcal{G}) \cap (\bigcup_{h=|L'|}^{l(\mathcal{G})} I_h)$ according to the following lemma.

Lemma 8: $C^d_{L'}(\mathcal{G}) \subseteq U^d_{L'}(\mathcal{G}) \cap \left(\bigcup_{h=|L'|}^{l(\mathcal{G})} I_h\right)$.

The index structure is basically the hierarchy of the vertices following $I_1, I_2, \ldots, I_{l(\mathcal{G})}$, that is, the vertices in I_i are placed on a lower level than those in I_{i+1}. Internally, the vertices in I_i are also placed on a stack of levels, which is determined as follows. Suppose the vertices in $I_1, I_2, \ldots, I_{i-1}$ have been removed from $\mathcal{G}$. Although the vertices $v \in I_i$ are iteratively removed from $\mathcal{G}$ due to $\mathsf{Num}(v) \leq i$, they are actually removed in different batches. In each batch, we select all the vertices v with $\mathsf{Num}(v) \leq i$ and remove them together. After a batch, some vertices v originally satisfying $\mathsf{Num}(v) > i$ may have $\mathsf{Num}(v) \leq i$ and thus will be removed in next batch. Therefore, in I_i, the vertices removed in the same batch are place on the same level, and the vertices removed in a later batch are placed on a higher level than the vertices removed in an early batch. In addition, let $L(v)$ be the set of layer numbers on which v is contained in the d-core just before v is removed from $\mathcal{G}$ in batch. We associate each vertex v in the index with $L(v)$. Moreover, we add an edge between vertices u and v in the index if (u, v) is an edge on a layer of $\mathcal{G}$.

By Lemma 8, we have narrowed down the search scope of $C^d_{L'}(\mathcal{G})$ from $U^d_{L'}(\mathcal{G})$ to $Z = U^d_{L'}(\mathcal{G}) \cap (\bigcup_{h=|L'|}^{l(\mathcal{G})} I_h)$. By exploiting the index, we can further narrow down the search scope. If there is no sequence of vertices $w_0, w_1, \ldots, w_n$ in the index such that $L' \subseteq L(w_0)$, $w_n = v$, w_i is on a higher level than w_{i+1}, and (w_i, w_{i+1}) is an edge in the index, then v is certainly not contained in $C^d_{L'}(\mathcal{G})$. The correctness of this method is guaranteed by the following lemma.

Lemma 9: For each vertex $v \in C^d_{L'}(\mathcal{G})$, there exists a sequence of vertices $w_0, w_1, \ldots, w_n$ in the index such that $L' \subseteq L(w_0)$, $w_n = v$, w_{i+1} is placed on a higher level than w_i, and (w_i, w_{i+1}) is an edge in the index.

Fast Search with Early Termination. Based on the index, Procedure RefineC in Fig. 10 searches for the exact $C^d_{L'}(\mathcal{G})$. First, we obtain the search scope $Z = U^d_{L'}(\mathcal{G}) \cap (\bigcup_{h=|L'|}^{l(\mathcal{G})} I_h)$ based on the index (line 1). By Lemma 8, we only need to consider the vertices in Z. Thus, before the search begins, we can remove all the vertices not in Z from the index (line 2).

Unlike Procedure dCC that only removes irrelevant vertices from $\mathcal{G}$, Procedure RefineC can find $C^d_{L'}(\mathcal{G})$ much faster by using two strategies: 1) Identify some vertices not in $C^d_{L'}(\mathcal{G})$ early; 2) Skip searching some vertices not in $C^d_{L'}(\mathcal{G})$. To this end, we set each vertex $v \in Z$ to one of the following three states: 1) v is *discarded* if it has been determined that $v \notin C^d_{L'}(\mathcal{G})$; 2) v is *undetermined* if v has been checked but has not be determined whether $v \in C^d_{L'}(\mathcal{G})$; 3) v is *unexplored* if it has not been checked by the search process. During the search process, a discarded vertex will not be involved in the following computation, and an undetermined vertex may become discarded due to the deletion of some edges. Initially, all vertices in Z are set to be unexplored (line 3).

```
Procedure RefineC(G, d, s, U^d_{L'}(G), L')
 1: Z = U^d_{L'}(G) ∩ (⋃_{h=|L'|}^{l(G)} I_h)
 2: removed all vertices not in Z from the index
 3: for each vertex v ∈ Z do
 4:     set all vertices in Z as unexplored
 5:     compute d_i^+(v) of all i ∈ L'
 6: for each level of the index do
 7:     if all vertices are unexplored or discarded on the level then
 8:         for each unexplored vertex v on the level do
 9:             if L' ⊈ L(v) then
10:                 set v as discarded
11:                 CascadeD(G, v, d, L')
12:             else
13:                 if v is not discarded then
14:                     set v as undetermined
15:                     for each unexplored neighbor u of v on a higher level do
16:                         set u as undetermined
17:     else
18:         for each undetermined vertex v on the level do
19:             if d_i^+(v) < d for some i ∈ L' then
20:                 set v as discarded
21:                 CascadeD(G, v, d, L')
22:             else
23:                 for each unexplored neighbor u of v on a higher level do
24:                     set v as undetermined
25:         for each unexplored vertex v on the level do
26:             set v as discarded
27:             CascadeD(G, v, d, L')
28: C^d_{L'}(G) ← {all undetermined vertices in Z}
29: return C^d_{L'}(G)
Procedure CascadeD(G, v, d, L')
 1: for each undetermined neighbor u of v do
 2:     d_i^+(u) ← d_i^+(u) − 1 for each i ∈ L' and (u, v) ∈ E_i(G)
 3:     if d_i^+(u) < d for some i ∈ L' then
 4:         set u as discarded
 5:         CascadeD(G, u, d, L')
```

Fig. 10. The RefineC Procedure.

For $i \in L'$, let $d_i^+(v)$ be the number of undetermined and unexplored vertices adjacent to v in $G_i[Z]$. Clearly, $d_i^+(v)$ is an upper bound on v's degree in $G_i[Z]$. If $d_i^+(v) < d$ on some layer $i \in L'$, we have $v \notin C^d_{L'}(\mathcal{G})$, so v is set as discarded. Notably, the removal of v may trigger the removal of other vertices. The details are shown in the CascadeD procedure. Specifically, if v is discarded, for each undetermined vertex $u \in Z$ adjacent to v, we decrease $d_i^+(u)$ by 1 if (u, v) is an edge on a layer $i \in L'$. If $d_i^+(u) < d$ for some $i \in L'$, we also set u as discarded and recursively invoke the CascadeD procedure to remove more vertices starting from u.

In the main search process, we check the vertices in Z in a level-by-level fashion. In each iteration (lines 6–27), we fetch all vertices on a level of the index and process them according to the following two cases:

Case 1 (lines 7–16): If there are only unexplored and discarded vertices on the current level, none of the vertices on this level has been checked before by the search process. At this point, we can check each unexplored vertex on this level. Specifically, for each unexplored vertex v, if $L' \not\subseteq L(v)$, we have $v \notin C^d_{L'}(\mathcal{G})$ by Lemma 9. Thus, we can immediately set v as discarded and invoke Procedure CascadeD to explore more discarded vertices starting from v (lines 10–11). Otherwise, if v is not discarded, we set v as undetermined (line 14). For each unexplored neighbor $u \in Z$ of v placed on a higher level than v in the index, we also set u as undetermined since u is possible to be contained in $C^d_{L'}(\mathcal{G})$ (line 16).

Algorithm TD-DCCS($\mathcal{G}, d, s, k$)
1: $\mathcal{G} \leftarrow$ VertexDeletion($\mathcal{G}, d, s$)
2: $\mathcal{R} \leftarrow$ InitTopK($\mathcal{G}, d, s, k$)
3: sort all layer numbers i in ascending order of $|C^d(G_i)|$, where $i \in [l(\mathcal{G})]$
4: construct the index of $\mathcal{G}$
5: $C^d_{[l(\mathcal{G})]} \leftarrow$ dCC($\mathcal{G}, [l(\mathcal{G})], d$)
6: TD-Gen($\mathcal{G}, d, s, k, [l(\mathcal{G})], C^d_{[l(\mathcal{G})]}, V(\mathcal{G}), \mathcal{R}$)
7: **return** $\mathcal{R}$

Fig. 11. The TD-DCCS Algorithm.

Case 2 (lines 17–27): If there is some undetermined vertices on the current level, we carry out the following steps. For each undetermined vertex v on this level, we check if $d_i^+(v) < d$ for some $i \in L'$ (line 19). If it is true, we have $v \notin C^d_{L'}(\mathcal{G})$. At this point, we set v to be discarded and invoke Procedure CascadeD to explore more discarded vertices starting from v (lines 20–21). Otherwise, v remains to be undetermined. For each unexplored neighbor $u \in Z$ of v placed on a higher level than v in the index, we also set u as undetermined since u is possible to be contained in $C^d_{L'}(\mathcal{G})$ (line 24).

For each vertex v that is still unexplored on the current level, none of the vertices in $C^d_{L'}(\mathcal{G})$ on lower levels than v in the index is adjacent to v. By Lemma 9, we have $v \notin C^d_{L'}(\mathcal{G})$. Thus, we can directly set v to be discarded and invoke Procedure CascadeD to explore more discarded vertices starting from v (lines 26–27).

After examining all levels in the index, $C^d_{L'}(\mathcal{G})$ is exactly the set of all undetermined vertices in Z (lines 28–29).

Time Complexity. Let $l' = |L'|$, $n' = U^d_{L'}(\mathcal{G})$, $m'_i = E_i[U^d_{L'}(\mathcal{G})]$ be the number of edges on layer i of the induced multi-layer graph $\mathcal{G}[U^d_{L'}(\mathcal{G})]$, and $m' = \sum_{i \in L'} m_i$. The following lemma shows that the time cost of the RefineC procedure is $O(n'l' + m')$. Notably, if we apply Procedure dCC on $\mathcal{G}[U^d_{L'}(\mathcal{G})]$ to find $C^d_{L'}(\mathcal{G})$ from scratch, the time cost is $O(n'l' + m''|L'|)$, where $m'' = |\bigcup_{i \in L'} E_i[U^d_{L'}(\mathcal{G})]|$. Since $m' \leq m''l'$ always holds, the time cost of Procedure RefineC is no more than Procedure dCC .

Lemma 10: The time complexity of Procedure RefineC is $O(n'l' + m')$.

D. Top-Down Algorithm

We present the complete top-down DCCS algorithm called TD-DCCS in Fig. 11. The input is a multi-layer graph $\mathcal{G}$ and parameters $d, s, k \in \mathbb{N}$. First, we apply the preprocessing methods of vertex deletion (line 1) and initializing of $\mathcal{R}$ (line 2). For the preprocessing method of sorting layers, we sort all layers i of $\mathcal{G}$ in ascending order of $|C^d(G_i)|$ at line 3 since a layer whose d-core is small is less likely to support a large d-CC. Then, we construct the index for $\mathcal{G}$ (line 4). Next, we invoke recursive Procedure TD-Gen to generate candidate d-CCs and update the result set $\mathcal{R}$ (line 6). Finally, $\mathcal{R}$ is returned as the result (line 7).

Theorem 4: The approximation ratio of TD-DCCS is $1/4$.

VI. Performance Evaluation

We implemented the proposed algorithms GD-DCCS, BU-DCCS and TD-DCCS in C++ and experimentally evaluated them in this section. We designate GD-DCCS as the baseline. Every algorithm is evaluated by its execution time (efficiency) and the cover size (accuracy). All the experiments were run on a machine with an Intel Core i5-2400 CPU (3.1GHz and 4 cores) and 22GB of RAM, running 64-bit Ubuntu 14.04.

Graph $\mathcal{G}$	$\lvert V(\mathcal{G})\rvert$	$\sum_{i=1}^{l(\mathcal{G})} \lvert E(G_i)\rvert$	$\lvert \bigcup_{i=1}^{l(\mathcal{G})} E(G_i)\rvert$	$l(\mathcal{G})$
PPI	328	4,745	3,101	8
Author	1,017	15,065	11,069	10
German	519,365	7,205,624	1,653,621	14
Wiki	1,140,149	7,833,140	3,309,592	24
English	1,749,651	18,951,428	5,956,877	15
Stack	2,601,977	63,497,050	36,233,450	24

Fig. 12. Statistics of Graph Datasets Used in Experiments.

Parameter	Range	Default Value
k	$\{5, 10, 15, 20, 25\}$	10
d	$\{2, 3, 4, 5, 6\}$	4
s (small)	$\{1, 2, 3, 4, 5\}$	3
s (large)	$\{l(\mathcal{G})-4, l(\mathcal{G})-3, l(\mathcal{G})-2, l(\mathcal{G})-1, l(\mathcal{G})\}$	$l(\mathcal{G})-2$
p	$\{0.2, 0.4, 0.6, 0.8, 1.0\}$	1.0
q	$\{0.2, 0.4, 0.6, 0.8, 1.0\}$	1.0

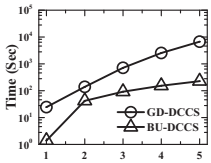
Fig. 13. Parameter Configuration.

Datasets. We use 6 real-world graph datasets of various types and sizes in the experiments. The statistics of the graph datasets are summarized in Fig. 12. *PPI* is a protein-protein interaction network extracted from the STRING database (`http://string-db.org`). It contains 8 layers representing the interactions between proteins detected by different methods. *Author* is a co-authorship network obtained from AMiner (`http://cn.aminer.org`). It contains 10 layers representing the collaboration between authors in 10 different years. *PPI* and *Author* are very small datasets. They are used in the comparisons between the notions of d-CC and quasi-clique. The other datasets were obtained from KONECT (`http://konect.uni-koblenz.de`) and SNAP (`http://snap.stanford.edu`), where each layer contains the connections generated in a specific time period. Specifically, in *German* and *English*, each layer consists of the interactions between users in a year; in *Wiki* and *Stack*, each layer contains the connections generated in an hour.

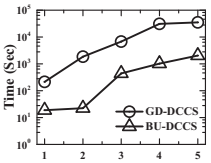
Parameters. We set 5 parameters in the experiments, namely k, d and s in the DCCS problem and $p, q \in [0, 1]$. Parameters p and q are varied in the scalability test of the algorithms. Specifically, p and q control the proportion of vertices and layers extracted from the graphs, respectively. The ranges and the default values of the parameters are shown in Fig. 13. We adopt two configurations for parameter s. When testing for small s, we select s from $\{1, 2, 3, 4, 5\}$; when testing for large s, we select s from $\{l(\mathcal{G})-4, l(\mathcal{G})-3, l(\mathcal{G})-2, l(\mathcal{G})-1, l(\mathcal{G})\}$. Without otherwise stated, when varying a parameter, the other parameters are set to their default values.

A. Experimental Results

Execution Time w.r.t. Parameter s. We evaluate the execution time of the algorithms w.r.t. s. First, we experiment for small s. Since TD-DCCS is not applicable when $s < l(\mathcal{G})/2$, we only test the other two algorithms for small s. Fig. 14 shows the execution time of the algorithms on the datasets *English* and *Stack*. We have two observations: 1) The execution time of all the algorithms substantially increases with s. This is simply because the search space of the DCCS problem fast grows with s when $s < l(\mathcal{G})/2$. 2) The BU-DCCS algorithm outperforms GD-DCCS by 1–2 orders of magnitude. For example, when $s = 4$, BU-DCCS is 39X and 30X faster than GD-DCCS on *English* and *Stack*, respectively. The main

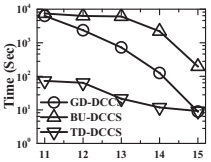


(a) English (Vary s)

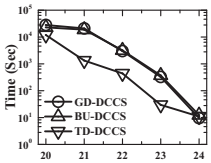


(b) Stack (Vary s)

Fig. 14. Execution Time vs Small s.

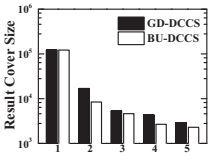


(a) English (Vary s)

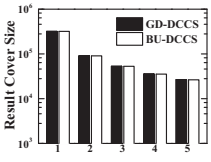


(b) Stack (Vary s)

Fig. 15. Execution Time vs Large s.

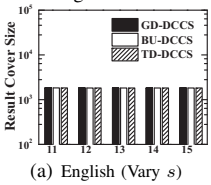


(a) English (Vary s)

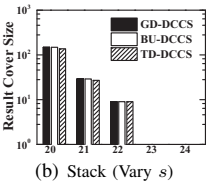


(b) Stack (Vary s)

Fig. 16. Result Cover Size vs Small s.

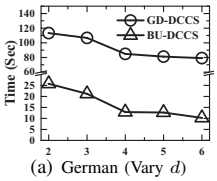


(a) English (Vary s)

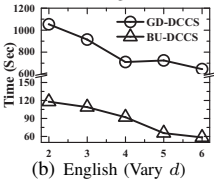


(b) Stack (Vary s)

Fig. 17. Result Cover Size vs Large s.

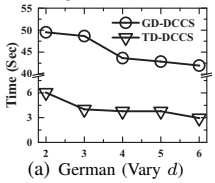


(a) German (Vary d)

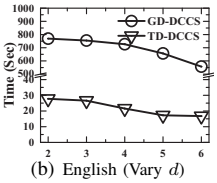


(b) English (Vary d)

Fig. 18. Execution Time vs d (Small s).



(a) German (Vary d)



(b) English (Vary d)

Fig. 19. Execution Time vs d (Large s).

reason is that the pruning techniques adopted by BU-DCCS reduce the search space of the DCCS problem by 80%–90%.

We also examine the algorithms for large s and show results in Fig. 15. At this time, we also test the TD-DCCS algorithm. We have the the following observations: 1) The execution time of all the algorithms decreases when s grows. This is because the search space of the DCCS problem decreases with s when $s \geq l(\mathcal{G})/2$. 2) BU-DCCS is not efficient for large s. Sometimes, it is even worse than GD-DCCS. When s is large, the sizes of the d-CCs significantly decrease. BU-DCCS has to search down deep the search tree until the pruning techniques start to take effects. In some cases, BU-DCCS searches even more d-CCs than GD-DCCS. 3) TD-DCCS runs much faster than all the others. For example, when $s = 13$, TD-DCCS is 50X faster than GD-DCCS on *English*. This is because d-CCs are generated in a top-down manner in TD-DCCS, so the number of d-CCs searched by TD-DCCS must be less than BU-DCCS. Moreover, many unpromising candidates d-CCs are pruned earlier in TD-DCCS.

Cover Size of Result w.r.t. Parameter *s*. We evaluate the cover size $|\mathsf{Cov}(\mathcal{R})|$ of the result $\mathcal{R}$ w.r.t. parameter s. Fig. 16 and Fig. 17 show the experimental results for small s and large s, respectively. We have two observations: 1) For all the algorithms, $|\mathsf{Cov}(\mathcal{R})|$ decreases with s. This is because while s increases, the size of a d-CC never increases due to Property 3, so $\mathcal{R}$ cannot cover more vertices. 2) In most cases, the results of the algorithms cover similar amount of vertices for either small s or large s. Sometimes, the result of GD-DCCS covers slightly more vertices than the results of BU-DCCS and TD-DCCS. This is because GD-DCCS is $(1 - 1/e)$-approximate; while BU-DCCS and TD-DCCS are $1/4$-approximate. It verifies that the practical approximation quality of BU-DCCS and TD-DCCS is close to GD-DCCS.

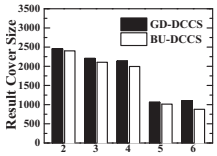
Effects of Parameter *d*. We examine the effects of parameter d on the performance of the algorithms. By varying d, Fig. 18 shows the execution time of BU-DCCS and GD-DCCS on datasets *German* and *English* for $s = 3$, and Fig. 19 shows the execution time of TD-DCCS and GD-DCCS on *German* and *English* for $s = l(\mathcal{G}) - 2$. We observe that the execution time of all the algorithms decreases as d grows. The reasons are as follows: 1) Due to Property 2, the size of a d-CC decreases as d grows. Thus, GD-DCCS takes less time in selecting d-CCs, and BU-DCCS and TD-DCCS take less time in updating temporary results. 2) While d increases, the size of the d-core on each layer decreases. By Lemma 1, the algorithms spend less time on d-CC computation. Moreover, both BU-DCCS and TD-DCCS are much faster than GD-DCCS.

Fig. 20 and Fig. 21 show the effects of d on the cover size of the results of BU-DCCS, TD-DCCS and GD-DCCS for small s and large s, respectively. We find that the cover size of the results decreases w.r.t. d for all the algorithms. This is simply because that the size of a d-CC decreases as d increases. Therefore, the results cover less vertices for larger d. Moreover, the practical approximation quality of BU-DCCS and TD-DCCS is close to GD-DCCS.

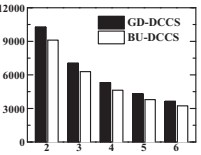
Effects of Parameter *k*. We examine the effects of parameter k on the performance of the algorithms. By varying k, Fig. 22 shows the execution time of BU-DCCS and GD-DCCS on datasets *Wiki* and *English* for $s = 3$, and Fig. 23 shows the execution time of TD-DCCS and GD-DCCS on *Wiki* and *English* for $s = l(\mathcal{G}) - 2$. We have the following observations: 1) The execution time of GD-DCCS increases with k because the time cost for selecting d-CCs in GD-DCCS is proportional to k. 2) Both BU-DCCS and TD-DCCS run much faster than GD-DCCS. 3) The execution time of BU-DCCS and TD-DCCS is insensitive to k. This is because the power of the pruning techniques in BU-DCCS and TD-DCCS relies on $|\mathsf{Cov}(\mathcal{R})|$ according to Eq. (1). As k grows, $|\mathsf{Cov}(\mathcal{R})|$ increases insignificantly, so k has little effects on the execution time of BU-DCCS and TD-DCCS.

Fig. 24 and Fig. 25 show the effects of k on the cover size of the results of BU-DCCS, TD-DCCS and GD-DCCS for small s and large s, respectively. We find that the cover size grows w.r.t. k; however, insignificantly for $k \geq 20$. From another perspective, it shows that there exists substantial overlaps among d-CCs. To reduce redundancy, it is meaningful to find top-k diversified d-CCs on a multi-layer graph.

Scalability w.r.t. Parameters *p* and *q*. We evaluate the scalability of the algorithms w.r.t. the input multi-layer graph size. We control the graph size by randomly selecting a fraction p of vertices or a fraction q of layers from the original graph. Note that we apply the small and large s when comparing BU-DCCS and TD-DCCS with GD-DCCS, respectively. Fig. 26 shows the execution time of BU-DCCS, TD-DCCS and GD-

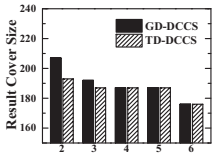


(a) German (Vary d)

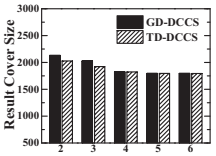


(b) English (Vary d)

Fig. 20. Result Cover Size vs d (Small s).

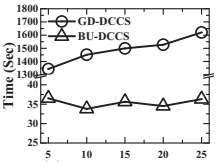


(a) German (Vary d)

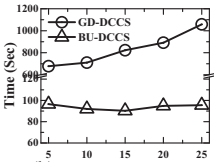


(b) English (Vary d)

Fig. 21. Result Cover Size vs d (Large s).

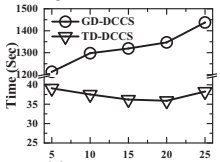


(a) Wiki (Vary k)

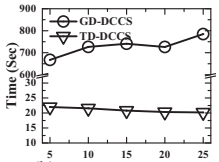


(b) English (Vary k)

Fig. 22. Execution Time vs k (Small s).

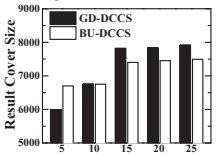


(a) Wiki (Vary k)

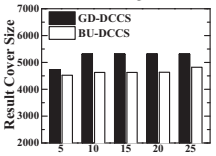


(b) English (Vary k)

Fig. 23. Execution Time vs k (Large s).

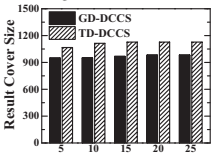


(a) Wiki (Vary k)

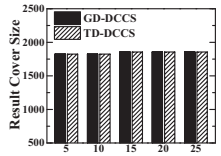


(b) English (Vary k)

Fig. 24. Result Cover Size vs k (Small s).

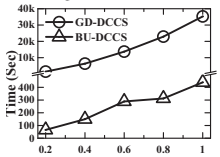


(a) Wiki (Vary k)

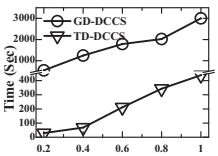


(b) English (Vary k)

Fig. 25. Result Cover Size vs k (Large s).

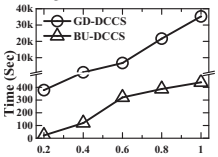


(a) Stack (Vary p, Small s)

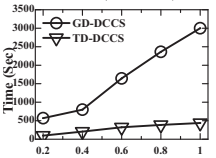


(b) Stack (Vary p, Large s)

Fig. 26. Execution Time vs p.

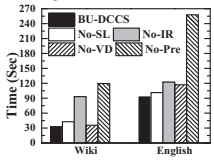


(a) Stack (Vary q, Small s)

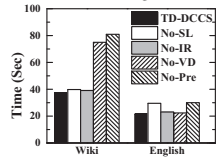


(b) Stack (Vary q, Large s)

Fig. 27. Execution Time vs q.



(a) Small s



(b) Large s

Fig. 28. Effects of Preprocessing.

DCCS on the largest dataset *Stack* by varying p from 0.2 to 1.0. All the algorithms scale linearly w.r.t. p because the time cost of computing d-CCs is linear to the vertex count.

Fig. 27 shows the execution time of BU-DCCS, TD-DCCS and GD-DCCS on *Stack* w.r.t. q. We observed that: 1) The execution time of all algorithms grows with q. This is simply because the search space of the DCCS problem increases when the input multi-layer graph contains more layers. 2) The execution time of GD-DCCS grows much faster than BU-DCCS and TD-DCCS. The main reason is that both BU-DCCS and TD-DCCS adopt the effective pruning techniques to significantly reduce the search space. The number of candidate d-CCs examined by GD-DCCS grows much faster than those examined by BU-DCCS and TD-DCCS.

Effects of Preprocessing Methods. We evaluate the effects of the preprocessing methods. Fig. 28 shows the comparison results for BU-DCCS and TD-DCCS. Here, No-VD, No-SL and No-IR mean the preprocessing method "vertex deletion", "sorting layers" and "result initialization" are disabled, respectively. No-Pre means all the preprocessing methods are disabled. We have the following observations: 1) Every preprocessing method can improve the efficiency of BU-DCCS and TD-DCCS. It verifies that the preprocessing methods can reduce the input graph size (by vertex deletion) and enhance the pruning power (by sorting layers and result initialization). 2) A preprocessing method may have different effects for different algorithms. For example, the result initialization method has more significant effects on BU-DCCS than on TD-DCCS. This is because for smaller s, the cover size of the result is much larger by Property 3. By Eq. (1), the initial result can eliminate more candidates d-CCs in BU-DCCS.

B. Comparison with Quasi-Clique-based Algorithms

We compare our algorithms with three representative quasi-clique-based algorithms, namely Crochet [11], Cocain [19] and MiMAG [4]. In general, these three algorithms use three parameters, γ, min_s and s, to constrain the properties of discovered densely connected vertex subsets Q on a multi-layer graph $\mathcal{G}$. Parameter $min_s \in \mathbb{N}$ specifies a *size constraint*: $|Q| \geq min_s$. Parameter $\gamma \in [0, 1]$ specifies a *density constraint*: Q must be a γ-*quasi-clique* on some layer G_i of $\mathcal{G}$, i.e., every vertex in $G_i[Q]$ has degree at least $\gamma(|Q| - 1)$. Parameter $s \in \mathbb{N}$ specifies a *support constraint*: Q must be a γ-quasi-clique on at least s layers of $\mathcal{G}$. Note that Crochet finds γ-quasi-cliques occurring on all layers of $\mathcal{G}$, so $s = l(\mathcal{G})$. For our algorithms, the density constraint is specified by parameter d. Since a d-CC contains at least $d+1$ vertices, our algorithms need not to set min_s. In terms of result redundancy, Crochet and Cocain return all results satisfying the constraints, while MiMAG and our algorithms find a set of diversified results.

On the same input, our BU-DCCS and TD-DCCS algorithms yield the same output in different time. For ease of presentation, we use DCCS to refer to the faster one.

Parameter Setting. 1) We set the same parameter s for all algorithms except Crochet ($s = l(\mathcal{G})$ for Crochet). 2) We specify the same parameters γ and min_s for all quasi-clique-based algorithms. 3) We independently set d and γ. 4) For the sake of fairness, we coordinate min_s with respect to d and γ. Specifically, min_s is set to the smallest integer such that $\lceil \gamma(min_s - 1) \rceil = d$. In this way, we have the same minimum degree constraint for all algorithms.

Execution Time. We test the execution time of the algorithms on the datasets *PPI*, *Author*, *German* and *Wiki* for $s = l(\mathcal{G})/2$, $d = 3$. We vary $\gamma = 0.5, 0.6$ and 0.8 and min_s is set accordingly to $6, 5$ and 4, respectively. Fig. 29 shows that DCCS $\prec$ Crochet $\prec$ MiMAG $\prec$ Cocain, where $\prec$ means "*faster than*". DCCS is 1–3 orders of magnitude faster than the other algorithms. This is because the search trees of BU-DCCS and TD-DCCS both contain $2^{l(\mathcal{G})}$ vertex subsets; however, the search trees of all quasi-clique-based algorithms contain $2^{|V(\mathcal{G})|}$ vertex subsets, where $l(\mathcal{G}) \ll |V(\mathcal{G})|$. Crochet

$\gamma = 0.5, d = 3, s = l(\mathcal{G})/2, min_s = 6$

Graph	Algorithm	Time (Sec)	Cover Size	Precision	Recall	F_1-Score
PPI	Crochet	4.97	29	0.403	1	0.574
	Cocain	31.66	87	0.764	0.618	0.683
	MiMAG	6.59	71	0.691	0.704	**0.699**
	DCCS	**0.05**	72	—	—	—
Author	Crochet	7.31	64	0.463	0.969	0.626
	Cocain	59.15	155	0.836	0.723	0.775
	MiMAG	23.86	131	0.858	0.878	**0.868**
	DCCS	**0.08**	134	—	—	—
German	Crochet	1,785.31	323	0.614	0.858	0.716
	Cocain	24,960.83	495	0.840	0.766	0.801
	MiMAG	2,479.81	413	0.778	0.850	**0.813**
	DCCS	**64.55**	451	—	—	—
Wiki	Crochet	3,691.77	851	0.254	0.917	0.398
	Cocain	68,206.13	3394	0.771	0.698	0.733
	MiMAG	29,173.68	2917	0.729	0.767	**0.747**
	DCCS	**667.33**	3072	—	—	—

$\gamma = 0.6, d = 3, s = l(\mathcal{G})/2, min_s = 5$

Graph	Algorithm	Time (Sec)	Cover Size	Precision	Recall	F_1-Score
PPI	Crochet	4.61	29	0.403	1	0.574
	Cocain	28.35	87	0.750	0.621	0.679
	MiMAG	6.26	67	0.685	0.731	**0.705**
	DCCS	**0.05**	72	—	—	—
Author	Crochet	7.23	64	0.440	0.922	0.596
	Cocain	54.86	154	0.836	0.727	0.778
	MiMAG	17.19	126	0.799	0.849	**0.823**
	DCCS	**0.08**	134	—	—	—
German	Crochet	1,766.18	315	0.596	0.854	0.702
	Cocain	20,851.09	479	0.827	0.779	0.802
	MiMAG	2,357.86	397	0.761	0.864	**0.809**
	DCCS	**64.55**	451	—	—	—
Wiki	Crochet	3301.21	824	0.248	0.924	0.391
	Cocain	67,351.74	3193	0.693	0.667	0.680
	MiMAG	27,945.16	2739	0.683	0.766	**0.722**
	DCCS	**667.33**	3072	—	—	—

$\gamma = 0.8, d = 3, s = l(\mathcal{G})/2, min_s = 4$

Graph	Algorithm	Time (Sec)	Cover Size	Precision	Recall	F_1-Score
PPI	Crochet	2.39	23	0.319	1	0.484
	Cocain	19.79	85	0.694	0.588	0.637
	MiMAG	5.93	59	0.653	0.796	**0.712**
	DCCS	**0.05**	72	—	—	—
Author	Crochet	7.19	62	0.440	0.952	0.602
	Cocain	51.21	137	0.784	0.766	0.775
	MiMAG	12.83	117	0.731	0.838	**0.781**
	DCCS	**0.08**	134	—	—	—
German	Crochet	1,037.54	291	0.574	0.890	0.698
	Cocain	19366.70	423	0.732	0.780	0.755
	MiMAG	2,298.74	382	0.711	0.840	**0.771**
	DCCS	**64.55**	451	—	—	—
Wiki	Crochet	2,953.98	793	0.246	0.952	0.391
	Cocain	62,831.70	2871	0.671	0.718	0.694
	MiMAG	21,897.50	2690	0.658	0.751	**0.701**
	DCCS	**667.33**	3072	—	—	—

Fig. 29. Comparison between DCCS and Quasi-Clique-based Algorithms Crochet, Cocain and MiMAG.

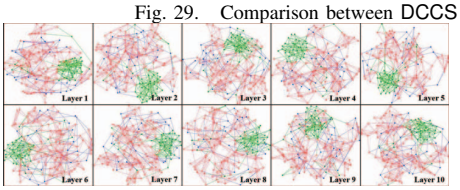


Fig. 30. Dense Subgraphs Found by DCCS and MiMAG on *Author*.

Metric	$d = 2$		$d = 3$		$d = 4$	
	MiMAG	DCCS	MiMAG	DCCS	MiMAG	DCCS
CCR	69.7%	**83.6%**	67.2%	**80.1%**	65.3%	**77.9%**
PCR	75.4%	**88.9%**	73.1%	**86.1%**	69.7%	**81.2%**
PWCR	63.3%	**74.9%**	60.5%	**71.6%**	54.2%	**66.8%**

Fig. 31. Comparison between DCCS and MiMAG in Application.

runs faster than MiMAG and Cocain because it finds quasi-cliques occurring on all layers rather than on s layers, and therefore more unpromising vertex subsets are pruned early. MiMAG is faster than Cocain because it only finds a set of diversified quasi-cliques, and thus many quasi-cliques with large overlaps are pruned earlier.

Comparison of Results. We also compare the results of the algorithms. Let $\mathcal{R}_D$, $\mathcal{R}_C$, $\mathcal{R}_N$ and $\mathcal{R}_M$ be the output of DCCS, Crochet, Cocain and MiMAG, respectively. For $\mathcal{R} \in \{\mathcal{R}_C, \mathcal{R}_N, \mathcal{R}_M\}$, we compare $\mathcal{R}_D$ with $\mathcal{R}$ by four measures: 1) *cover sizes* $|\mathsf{Cov}(\mathcal{R}_D)|$ and $|\mathsf{Cov}(\mathcal{R})|$; 2) *precision* $\frac{|\mathsf{Cov}(\mathcal{R}_D) \cap \mathsf{Cov}(\mathcal{R})|}{|\mathsf{Cov}(\mathcal{R}_D)|}$; 3) *recall* $\frac{|\mathsf{Cov}(\mathcal{R}_D) \cap \mathsf{Cov}(\mathcal{R})|}{|\mathsf{Cov}(\mathcal{R})|}$; 4) F_1*-score*, the *harmonic average* of precision and recall. As shown in Fig. 29, we have the following observations.

1) DCCS ($\mathcal{R}_D$) vs. Corchet ($\mathcal{R}_C$). We find that $\mathcal{R}_D$ covers more vertices than $\mathcal{R}_C$. According to the *recall* measure, 85%–100% vertices covered by $\mathcal{R}_C$ are also covered by $\mathcal{R}_D$. This is because Corchet finds quasi-cliques on all layers, which are highly likely to be covered by the d-CCs found by DCCS.

2) DCCS ($\mathcal{R}_D$) vs. Cocain ($\mathcal{R}_N$). The vertices covered by $\mathcal{R}_D$ significantly overlaps with the vertices covered by $\mathcal{R}_N$. According to the precision, 67%–78% vertices covered by $\mathcal{R}_D$ are also covered by $\mathcal{R}_N$; according to the recall, 59%–78% vertices covered by $\mathcal{R}_N$ are also covered by $\mathcal{R}_D$.

3) DCCS ($\mathcal{R}_D$) vs. MiMAG ($\mathcal{R}_M$). The vertices covered by $\mathcal{R}_D$ largely overlaps with the vertices covered by $\mathcal{R}_M$: 65%–85% vertices covered by $\mathcal{R}_D$ are covered by $\mathcal{R}_M$; in reverse, 70%–88% vertices covered by $\mathcal{R}_M$ are covered by $\mathcal{R}_D$.

4) MiMAG ($\mathcal{R}_M$) vs. Cocain ($\mathcal{R}_N$) vs. Corchet ($\mathcal{R}_C$). According to the F_1-score, the output of DCCS is closer to $\mathcal{R}_M$ and $\mathcal{R}_N$ than to $\mathcal{R}_C$. Since Cocain returns all qualified results, while MiMAG returns diversified results, $\mathcal{R}_M$ is more concise than $\mathcal{R}_N$, and MiMAG is much faster than Cocain. Therefore, in terms of both quality and efficiency, MiMAG outperforms Cocain and Corchet. For this reason, we present more details on the comparison results between DCCS and MiMAG.

DCCS vs. MiMAG (Visualization). To better understand the differences between $\mathcal{R}_D$ and $\mathcal{R}_M$, we visualize $\mathcal{R}_D$ and $\mathcal{R}_M$ on the small dataset *Author*. For parameters $s = 5$, $d = 3$, $\gamma = 0.8$ and $min_s = 4$, Fig. 30 shows the subgraphs induced by $\mathsf{Cov}(\mathcal{R}_D)$ and $\mathsf{Cov}(R_M)$ on all the ten layers. The vertices in $\mathsf{Cov}(\mathcal{R}_D) \cap \mathsf{Cov}(\mathcal{R}_M)$, $\mathsf{Cov}(\mathcal{R}_D) - \mathsf{Cov}(\mathcal{R}_M)$ and $\mathsf{Cov}(\mathcal{R}_M) - \mathsf{Cov}(\mathcal{R}_D)$ are colored in red, green and blue, respectively. We have two observations: 1) The (blue) vertices in $\mathsf{Cov}(\mathcal{R}_M) - \mathsf{Cov}(\mathcal{R}_D)$ are sparsely connected compared with the (red) vertices in $\mathsf{Cov}(\mathcal{R}_M) \cap \mathsf{Cov}(\mathcal{R}_D)$. 2) The (green) vertices in $\mathsf{Cov}(\mathcal{R}_D) - \mathsf{Cov}(\mathcal{R}_M)$ are densely connected with themselves and the (red) vertices in $\mathsf{Cov}(\mathcal{R}_D) \cap \mathsf{Cov}(\mathcal{R}_M)$; however, the dense subgraph induced by the green vertices is not found by MiMAG.

DCCS vs. MiMAG (Application). The dataset *PPI* contains eight *protein-protein interaction networks* of *W. Glossinidia* detected by different methods. As an application, we use DCCS and MiMAG to find dense subgraphs on the dataset *PPI*, which are closely related to *protein complexes*. We take the protein complexes of *W. Glossinidia* recorded in the MIPS database (`http://mips.helmholtz-muenchen.de`) as the ground truth. To evaluate the results of DCCS and MiMAG, we introduce three measures: 1) *Complete Containment Ratio (CCR)*: the ratio of protein complexes completely contained in the result; 2) *Partial Containment Ratio (PCR)*: the ratio of protein complexes partially (more than 60%) contained in the result; 3) *Pair-wise Containment Ratio (PWCR)*: the ratio of protein pairs in the result co-existing in a known protein complex. Fig. 31 shows the experimental results for $2 \leq d \leq 4$. We have the following observations. 1) In terms of all the measures, DCCS outperforms MiMAG. This is because the dense subgraphs found by DCCS cover more densely connected vertices than MiMAG. 2) As d increases, the values of all measures decrease. This is because as d increases, the results of DCCS and MiMAG both cover less vertices.

Summary. Cross-graph quasi-clique and d-CC are two different notions to represent dense subgraphs on multi-layer graphs. The quasi-clique-base algorithms find multiple cohesively connected vertices in small diameters. Our DCCS algorithms find large-diameter d-CCs. Our algorithms also run much faster than the quasi-clique-based algorithms and can find dense subgraphs missed by the quasi-clique-based algorithms.

VII. Related Work

Recently, Kim and Lee [8] surveyed dense subgraph mining on graphs with multiple types of edges. Here, we briefly review the existing work that is most related to our work.

Dense Subgraph Mining on Two-layer Graphs. Two-layer graph is a special case of multi-layer graph, in which two lay-

ers represents physical links and conceptual links, respectively. The dense subgraph mining algorithms take both physical and conceptual connections into account. Ref. [10] proposed an algorithm to find dense subgraphs by expanding from seed vertices. Ref. [12] proposed an algorithm which adopts edge-induced matrix factorization. In [20], structures and attributes are combined to form a unified distance measure, and a clustering algorithm is applied to detect dense subgraphs. In [17], structures and attributes are fused by a probabilistic model, and a model-based algorithm is proposed to find dense subgraphs. Other methods for finding dense subgraphs on two-layer graphs are based on correlated pattern mining [14] and graph merging [13]. All these algorithms are tailored to fit two-layer graphs with a physical layer and a conceptual layer. They cannot be adapted to process general multi-layer graphs.

Dense Subgraph Mining on General Multi-layer Graphs. A general multi-layer graph is composed by several layers representing various types of edges. The work [5][16] find dense subgraphs on multi-layer graphs by matrix factorization. The goal is to approximate the adjacency matrix and the Laplacian matrix of the graph on each layer. However, the matrix-based methods require huge amount of memory and thus are not scalable to large graphs. The work [4][11][19] extends the quasi-clique notion defined on single-layer graphs. In [19][11], the algorithms find all cross-graph quasi-cliques; in [4], diversified quasi-cliques are found to reduce redundancy. However, as discussed in Section I, the quasi-clique-based methods are computationally costly, and the diameter of the discovered quasi-cliques are often very small. As verified by the experimental results in Section VI, the quasi-clique-based methods may miss useful results.

Frequent Subgraph Pattern Mining. Given a set D of labelled graphs, frequent subgraph pattern mining discovers all subgraph patterns that are subgraph isomorphic to at least a fraction *minsup* of graphs in D [18]. Our work is different from frequent subgraph pattern mining due to the following reason. A frequent subgraph pattern represents a common substructure recurring in many graphs in D. However, a d-CC is a set of vertices, and they may not induce the same interlink structure on different layers of a multi-layer graph.

Clustering on Heterogeneous Information Networks. Heterogeneous information network (HIN) is a logical network composed by multiple types of links between multiple types of objects. The clustering problem on HINs has been well studied in [15]. This work is different from our work in two aspects: 1) HIN characterizes the relationships between different types of objects. Normally, only one type of edges between two types of vertices are considered. However, a multi-layer graph models multiple types of relationships between homogenous objects of the same type. 2) HIN is single-layer graph. The clustering algorithm only consider the cohesiveness of a vertex subset rather than its support.

VIII. Conclusions

This paper addresses the diversified coherent core search (DCCS) problem on multi-layer graphs. The new notion of d-coherent core (d-CC) has three elegant properties, namely uniqueness, hierarchy, and containment. The greedy algorithm is $(1-1/e)$-approximate; however, it is not efficient on large multi-layer graphs. The bottom-up and the top-down DCCS algorithms are $1/4$-approximate. For $s < l(\mathcal{G})/2$, the bottom-up algorithm is faster than the other ones; for $s \geq l(\mathcal{G})/2$, the top-down algorithm is faster than the other ones. The DCCS algorithms outperform the quasi-clique-based cohesive subgraph mining algorithms in terms of both time efficiency and result quality.

Acknowledgements. This work was partially supported by the National Natural Science Foundation of China under Grant No. 61672189, 61532015 and 61732003.

References

[1] A. Angel, N. Koudas, N. Sarkas, D. Srivastava, M. Svendsen, and S. Tirthapura. Dense subgraph maintenance under streaming edge weight updates for real-time story identification. *PVLDB*, 5(6):574–585, 2012.

[2] G. Ausiello, N. Boria, A. Giannakos, G. Lucarelli, and V. T. Paschos. Online maximum k-coverage. In *International Conference on Fundamentals of Computation Theory*, pages 181–192, 2011.

[3] V. Batagelj and M. Zaversnik. An o(m) algorithm for cores decomposition of networks. *Computer Science*, 1(6):34–37, 2003.

[4] B. Boden, S. Nnemann, H. Hoffmann, and T. Seidl. Mining coherent subgraphs in multi-layer graphs with edge labels. In *KDD*, pages 1258–1266, 2012.

[5] X. Dong, P. Frossard, P. Vandergheynst, and N. Nefedov. Clustering with multi-layer graphs: A spectral perspective. *IEEE Transactions on Signal Processing*, 60(11):5820–5831, 2011.

[6] T. Frickey and G. Weiller. Mclip: motif detection based on cliques of gapped local profile-to-profile alignments. *Bioinformatics*, 23(4):502–3, 2007.

[7] H. Hu, X. Yan, Y. Huang, J. Han, and X. J. Zhou. Mining coherent dense subgraphs across massive biological networks for functional discovery. *Bioinformatics*, 21(suppl_1):i213, 2005.

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

[9] V. E. Lee, N. Ruan, R. Jin, and C. C. Aggarwal. A survey of algorithms for dense subgraph discovery. In *Managing and Mining Graph Data*, pages 303–336. Springer, 2010.

[10] H. Li, Z. Nie, W. C. Lee, L. Giles, and J. R. Wen. Scalable community discovery on textual data with relations. In *CIKM*, pages 1203–1212, 2008.

[11] J. Pei, D. Jiang, and A. Zhang. On mining cross-graph quasi-cliques. In *KDD*, pages 228–238, 2005.

[12] G. J. Qi, C. C. Aggarwal, and T. Huang. Community detection with edge content in social media networks. In *ICDE*, pages 534–545, 2012.

[13] Y. Ruan, D. Fuhry, and S. Parthasarathy. Efficient community detection in large networks using content and links. In *WWW*, pages 1089–1098, 2012.

[14] A. Silva, W. M. Jr, and M. J. Zaki. Mining attribute-structure correlated patterns in large attributed graphs. *PVLDB*, 5(5):466–477, 2012.

[15] Y. Sun, Y. Yu, and J. Han. Ranking-based clustering of heterogeneous information networks with star network schema. In *KDD*, pages 797–806, 2009.

[16] W. Tang, Z. Lu, and I. S. Dhillon. Clustering with multiple graphs. In *ICDM*, pages 1016–1021, 2009.

[17] Z. Xu, Y. Ke, Y. Wang, H. Cheng, and J. Cheng. A model-based approach to attributed graph clustering. In *SIGMOD*, pages 505–516, 2012.

[18] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *ICDM*, pages 721 – 724, 2002.

[19] Z. Zeng, J. Wang, L. Zhou, and G. Karypis. Coherent closed quasi-clique discovery from large dense graph databases. In *KDD*, pages 797–802, 2006.

[20] Y. Zhou, H. Cheng, and J. X. Yu. Graph clustering based on structural/attribute similarities. *PVLDB*, 2(1):718–729, 2009.

[21] R. Zhu, Z. Zou, and J. Li. Diversified coherent core search on multi-layer graphs. *http://arxiv.org/abs/1709.09471*, 2017.