The VLDB Journal (2019) 28:597–622
https://doi.org/10.1007/s00778-019-00542-3

REGULAR PAPER

Fast diversified coherent core search on multi-layer graphs

Rong Zhu[1] · **Zhaonian Zou**[1] · **Jianzhong Li**[1]

Received: 12 March 2018 / Revised: 15 March 2019 / Accepted: 15 May 2019 / Published online: 1 July 2019

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. Furthermore, we propose some optimization techniques to further speed up the algorithms. 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.

Keywords Multi-layer graph · Dense subgraph · Coherent core · Diversity

1 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 [16], 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 [13,27]; in social networks, users can interact through different social media [21]. In [5,20], such a graph with multiple types of edges is modeled as a *multi-layer graph*, where each layer independently accommodates a certain type of edges.

✉ Zhaonian Zou
znzou@hit.edu.cn

Rong Zhu
rzhu@hit.edu.cn

Jianzhong Li
lijzh@hit.edu.cn

[1] School of Computer Science and Technology, Harbin Institute of Technology, Harbin, Heilongjiang Province, China

Finding dense subgraphs on multi-layer graphs has witnessed many real-world applications. We show two examples as follows.

Application 1 (joint mining of biological modules) Biological networks represent interactions between proteins and genes. These interactions can be detected by several different methods such as biological experiments, co-expression, gene co-occurrence, and text mining [27]. Finding groups of genes and proteins cohesively interacting with each other, called biological modules, is interesting and important in bioinformatics [13,20]. However, the interaction data obtained by a certain method are usually very noisy, which makes the detection results unconvincing or unreliable [20]. To filter out the effects of spurious interactions, biologists try to jointly analyze the interactions in a collection of biological networks. In particular, they model these biological networks as a multi-layer graph, where each layer contains all interactions (edges) detected by a certain method. A set of vertices is regarded as a reliable biological module if they are simultaneously densely connected across multiple layers [20].

Application 2 (extracting active co-author groups) Co-authorship networks such as DBLP represent collaboration between authors. Following active co-author groups that frequently occur in research communities helps learn more about research trends and hot spots in research domains. To extract active co-author groups, scientists often organize co-authorship networks as multi-layer graphs. All connections

between authors are categorized into different layers either based on time periods [31] or based on conferences [5]. A group of authors are active if they have close collaborations in multiple time periods or in multiple conferences.

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* [5,20,32]. 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 [20], 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 [5].
(2) Quasi-cliques are useful in the study of micro-clusters (e.g., motifs [12]) 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-processing [16]. 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 four-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:

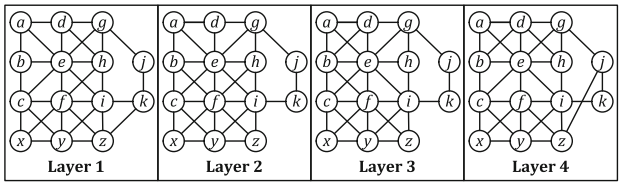


Fig. 1 Example of four-layer graph

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 [5], 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. Reference [5] 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 [2,5], 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 $1/4$.

To further speed up the algorithms for the DCCS problem, we develop some optimization techniques. We introduce an index structure, which organizes all the vertices hierarchically. Base on this index, we propose a faster d-CC computation method with less examination of vertices. The faster d-CC computation method can be applied to all the proposed algorithms.

We conducted extensive experiments on a variety of real-world 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 optimized greedy, bottom-up, and top-down algorithms run faster than the original ones, respectively.
(3) The practical approximation quality of the bottom-up and top-down algorithms is comparable to that of the greedy algorithm.
(4) Our DCCS algorithms outperform the cross-graph quasi-clique mining algorithms [5,20,32] on multi-layer graphs in terms of both execution time and result quality.

The rest of the paper is organized as follows. Section 2 introduces the basic concepts and formalizes the DCCS problem. Section 3 introduces a method for computing d-CCs. Section 4 presents the greedy algorithm. The bottom-up and the top-down search algorithms are described in Sects. 5 and 6, respectively. The optimized algorithms are proposed in Sect. 7. The experimental results are reported in Sect. 8. Section 9 reviews the related work, and Sect. 10 concludes this paper.

2 Problem definition

In this section, we introduce some basic concepts and formalize the problem.

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 the tuple $(V, E_1, E_2, \ldots, E_l)$, 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 we have $C^d(G) \subseteq C^{d-1}(G) \subseteq \cdots \subseteq C^1(G) \subseteq C^0(G)$ 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 three properties.

Symbols and notations related to basic concepts (in Sect. 2)	
G	Single-layer graph
$\mathcal{G}$	Multi-layer graph
u, v, w	Vertex
(u, v)	Edge
$V(G)(V(\mathcal{G}))$	The vertex set of single-layer graph G (multi-layer $\mathcal{G}$)
$G[S](\mathcal{G}[S])$	The subgraph of single-layer graph G (multi-layer graph $\mathcal{G}$) induced by vertex subset S
$E_i(\mathcal{G})$, $E(G_i)$	The edge set on layer i of multi-layer graph $\mathcal{G}$
$l(\mathcal{G})$	The number of layers in multi-layer graph $\mathcal{G}$
$[n]$	The set $\{1, 2, \ldots, n\}$
$d_G(v)$	The degree of vertex v in single-layer graph G
$C^d(G)$	The d-core on single-layer graph G
$C^L_d(\mathcal{G})$	The d-CC on multi-layer graph $\mathcal{G}$ w.r.t. layer subset L
Symbols and notations related to the DCCS *problem (in Sect. 2)*	
d	Degree threshold
k	The desired number of diversified d-CCs
s	Support threshold
$\mathcal{F}_{d,s}(\mathcal{G})$, $\mathcal{F}$	The set of d-CCs on s layers of multi-layer graph $\mathcal{G}$
$\mathcal{R}$	The result set of diversified d-CCs
$\mathsf{Cov}(\mathcal{R})$	The cover set of $\mathcal{R}$, that is, $\bigcup_{C\in\mathcal{R}} C$
Symbols and notations related to the algorithms (in Sects. 5 and 6)	
$C^*(\mathcal{R})$ (in Sect. 5.1)	The d-CC in $\mathcal{R}$ that covers the minimum number of vertices exclusively by itself
$U^d_L(\mathcal{G})$ (in Sect. 6.1)	The potential vertex set w.r.t. the d-CC $C^d_L(G)$
$M_L(N_L)$ (in Sect. 6.2)	The set of layer numbers in Class 1 (Class 2) in L
I (in Sect. 6.3)	The index of the multi-layer graph

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 $d \in \mathbb{N}$.*

For readability, we put the some proofs of properties, lemmas and theorems in Appendix A of this paper.

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 [2,5], we assess the diversity of the discovered d-CCs by the number of vertices they cover and try to maximize the diversity

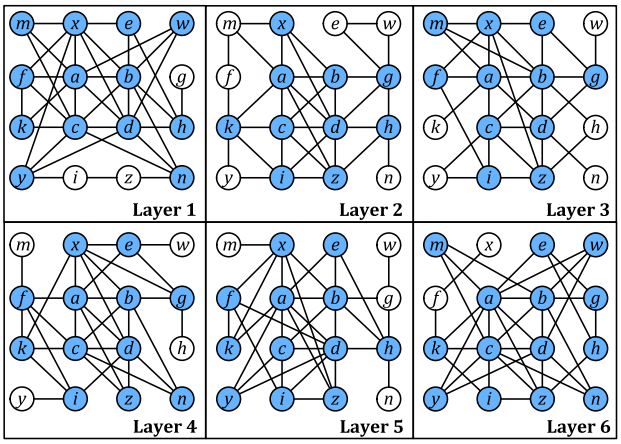


Fig. 2 Example of a six-layer graph

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 $d \in \mathbb{N}$, a minimum support threshold $s \in \mathbb{N}$, and the number $k \in \mathbb{N}$ 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.*

Figure 2 shows a six-layer graph $\mathcal{G}$. Let $d = 3$, $s = 2$, and $k = 2$. The d-cores on all layers are highlighted in blue. The result of the DCCS problem is $\mathcal{R} = \{C^d_{\{1,6\}}(\mathcal{G}), C^d_{\{4,5\}}(\mathcal{G})\}$, where $C^d_{\{1,6\}}(\mathcal{G}) = \{a, b, c, d, n, w, y\}$ and $C^d_{\{4,5\}}(\mathcal{G}) = \{a, b, c, d, f, i, k, x, z\}$. We have $|\mathsf{Cov}(\mathcal{R})| = 12$.

3 The *d*-CC computation algorithm

In this section, we propose an algorithm for finding $C^d_L(\mathcal{G})$, the d-CC in a multi-layer graph $\mathcal{G}$ w.r.t. a set L of layer numbers. This algorithm is a key component of the algorithms described in the next sections for the DCCS problem.

Our d-CC computation algorithm is inspired by the d-core decomposition algorithm [3]. In this d-core decomposition algorithm, the vertices whose degrees are less than d are iteratively removed from the input graph. Finally, the remaining vertices form the d-core of the graph. Our d-CC computation algorithm follows a similar paradigm. By the definition of d-CC, each vertex v in $C^d_L(\mathcal{G})$ is adjacent to at least d vertices in $C^d_L(\mathcal{G})$ on each layer in L. According to this fact, the central idea of our algorithm for computing $C^d_L(\mathcal{G})$ is removing from $\mathcal{G}$ the vertices that cannot satisfy this degree constraint. Specifically, let $m(v) = \min_{i\in L} d_{G_i}(v)$ be the minimum degree of vertex v on all layers in L. If $m(v) < d$,

Procedure dCC($\mathcal{G}, d, L$)
1: compute $m(v)$ for all vertices v of $\mathcal{G}$
2: initialize four arrays ver, pos, sbin and bin
3: $M \leftarrow \max_{v \in V(\mathcal{G})} m(v)$
4: **for** each vertex $v \in V(\mathcal{G})$ **do**
5: sbin$[m(v)] \leftarrow$ sbin$[m(v)] + 1$
6: bin$[0] \leftarrow 0$
7: **for** $i \leftarrow 1$ to M **do**
8: bin$[i] \leftarrow$ bin$[i-1] +$ sbin$[i-1]$
9: bin$' \leftarrow$ bin
10: **for** each vertex $v \in V(\mathcal{G})$ **do**
11: ver[bin$'[m(v)]] \leftarrow v$
12: pos$[v] \leftarrow$ bin$'[m(v)]$
13: bin$'[m(v)] \leftarrow$ bin$'[m(v)] + 1$
14: **repeat**
15: $v \leftarrow$ the first vertex in the array ver
16: **if** $m(v) < d$ **then**
17: delete v from the array ver
18: remove v and its incident edges from all layers of $\mathcal{G}$
19: **for** each vertex u adjacent to v on some layers **do**
20: recompute $m(u)$ as $m(u)'$
21: **if** $m(u) \neq m(u)'$ **then**
22: $w \leftarrow$ ver[bin$[m(u)]]$
23: $pw \leftarrow$ pos$[w]$; $pu \leftarrow$ pos$[u]$
24: ver$[u] \leftarrow w$; ver$[w] \leftarrow u$
25: pos$[u] \leftarrow pw$; pos$[w] \leftarrow pu$
26: bin$[m(u)] \leftarrow$ bin$[m(u)] + 1$
27: **until** $m(v) \geq d$
28: **return** $V(\mathcal{G})$

Fig. 3 The dCC procedure

the degree of v must be less than d on a certain layer in L, so we have $v \notin C_L^d(\mathcal{G})$ and thus remove v from $\mathcal{G}$. Notice that removing v may cause some vertices adjacent to v on some layers in L not satisfy the degree constraint any more. Hence, we *iteratively* remove all such irrelevant vertices from $\mathcal{G}$ until $m(v) \geq d$ for all vertices v remaining in $\mathcal{G}$. Finally, the set of vertices remaining in $\mathcal{G}$ is identical to $C_L^d(\mathcal{G})$.

In the following, we present an efficient implementation of the d-CC computation method. The dCC procedure in Fig. 3 describes the pseudocode of this algorithm. The procedure takes as input a multi-layer graph $\mathcal{G}$, an integer $d \in \mathbb{N}$, and a subset $L \subseteq [l(\mathcal{G})]$ and outputs the d-CC $C_L^d(\mathcal{G})$ in linear time w.r.t. the size of $\mathcal{G}$. It works as follows. At the beginning, we compute $m(v)$ for all vertices $v \in V(\mathcal{G})$ (line 1). In the dCC procedure, we scan all vertices of $\mathcal{G}$ only once and update $m(v)$ when needed. To this end, we exploit four arrays, namely ver, pos, sbin, and bin.

- The array ver stores all vertices v of $\mathcal{G}$ in the ascending order of $m(v)$. In ver, the consecutive vertices having the same value of $m(v)$ constitute a *bin* in ver.
- The array pos records the position (offset) of each vertex v of $\mathcal{G}$ in the array ver, i.e., ver[pos$[v]] = v$.
- The array sbin records the size of each bin in ver, i.e., sbin$[i]$ is the number of vertices v such that $m(v) = i$.
- The array bin records the starting position (offset) of each bin in ver, i.e., bin$[i]$ is the position of the first vertex v in ver such that $m(v) = i$.

Lines 2–13 set up the arrays. Let $M = \max_{v \in V(\mathcal{G})} m(v)$. Obviously, the respective size of sbin and bin is $M + 1$. First, we scan all vertices in $V(\mathcal{G})$ to determine the size of each bin (lines 4–5). Then, we accumulate the bins' size in sbin to obtain the starting position of each bin (lines 6–8). Based on the starting positions, we scan all vertices of $\mathcal{G}$ and place them into ver and pos accordingly (lines 9–13).

In the main iterations (lines 14–27), each time we check the first vertex v in the array ver (line 15). If $m(v) < d$, we have $v \notin C_L^d(\mathcal{G})$, so we delete v from the array ver (line 17) and remove v and its incident edges from all layers of $\mathcal{G}$ (line 18). Then, for each vertex u adjacent to v on some layers, $m(u)$ may change. Note that $m(u)$ can be decreased by at most 1 since we remove only one neighbor of u from $\mathcal{G}$. Therefore, we recompute $m(u)$ as $m(u)'$ (line 20). If $m(u)' \neq m(u)$, we move vertex u from the $m(u)$th bin to the $(m(u)-1)$th bin. Let w be the first vertex in the $m(u)$th bin (line 22). We can exchange the positions of u and w in the array ver and exchange pos$[w]$ and pos$[v]$ accordingly (lines 23–25). After that, bin$[m(u)]$ is increased by 1 to indicate that u is removed from its bin (line 26).

When the iterations terminate, all the vertices remaining in $\mathcal{G}$ are returned as $C_L^d(\mathcal{G})$ (line 28). The correctness of the dCC procedure is obvious.

Complexity analysis Let $n = |V(\mathcal{G})|$, $l = l(\mathcal{G})$, $m_i = |E_i(\mathcal{G})|$, and $m = |\bigcup_{i \in [l(\mathcal{G})]} E_i(\mathcal{G})|$. Line 1 computes $m(v)$ for all vertices $v \in V(\mathcal{G})$ in $O(nl)$ time. Lines 2–13 set up elements in the four arrays in $O(n)$ time. In the main iterations, for each vertex u, the time for updating $m(u)$ and changing the position of u is $O(l)$. Let $d_{\mathcal{G}}(u)$ denote $|\bigcup_{i \in [l(\mathcal{G})]} N_{G_i}(u)|$. The vertex u can be accessed by its neighbors at most $d_{\mathcal{G}}(u)$ times. Thus, the total time cost of the main iterations is at most $O(\sum_{u \in V(\mathcal{G})} d_{\mathcal{G}}(u) l) = O(2ml)$. Consequently, the time complexity of dCC is $O(nl + n + 2ml) = O(nl + ml)$. The dCC procedure needs $O(n)$ extra space to store the three arrays and $m(v)$ of each vertex $v \in V(\mathcal{G})$. Thus, the space complexity of dCC is $O(n)$.

4 The greedy algorithm

A straightforward solution to the DCCS problem is generating all candidate d-CCs and selecting k d-CCs that cover the maximum number of vertices. However, the search space of all k-combinations of d-CCs is extremely large, so this method is intractable even for small multi-layer graphs. Alternatively, fast algorithms with provable performance guarantees may be more preferable. 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 $C_{L_1 \cup L_2}^d(\mathcal{G}) \subseteq C_{L_1}^d(\mathcal{G}) \cap C_{L_2}^d(\mathcal{G})$ 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$(\mathcal{G}[S], d, L)$
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** $\mathcal{R}$

Procedure VertexDeletion$(\mathcal{G}, d, s)$
1: **repeat**
2: **for** $i \leftarrow 1$ **to** $l(\mathcal{G})$ **do**
3: compute the d-core $C^d(G_i)$ on graph G_i
4: **for** each $v \in V(\mathcal{G})$ **do**
5: **if** Supp$(v) < s$ **then**
6: remove v and its incident edges from all layers of $\mathcal{G}$
7: **until** Supp$(v) \geq s$ for all $v \in V(\mathcal{G})$
8: **return** $\mathcal{G}$

Fig. 4 The GD-DCCS algorithm

The greedy algorithm GD-DCCS is described in Fig. 4. 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).

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 $\mathcal{R}$, and remove C^* from $\mathcal{F}$. Finally, $\mathcal{R}$ is output as the result (line 12).

Complexity analysis 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 $O(ms)$ time as shown in Sect. 3. 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* $1 - \frac{1}{e}$.

Proof At lines 1–8, GD-DCCS exactly finds $\mathcal{F}$, the set of all candidate d-CCs. The remaining part of GD-DCCS aims at finding the set $\mathcal{R}$ of k d-CCs from $\mathcal{F}$ that maximizes $|\mathsf{Cov}(\mathcal{R})|$. This is an instance of the *max-k-cover problem* [2]. Lines 9–11 of GD-DCCS actually use the greedy algorithm [2] to solve this problem. The approximation ratio of this greedy algorithm is $1 - 1/e$ [2]. Thus, the theorem holds. □

Vertex deletion procedure In line 1 of GD-DCCS, we apply Procedure VertexDeletion to remove some unpromising vertices from $\mathcal{G}$. Let Supp(v) denote the number of layers i such that $v \in C^d(G_i)$. If Supp$(v) < s$, 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 Supp$(v) \geq s$ for all vertices v remaining in $\mathcal{G}$.

The VertexDeletion procedure takes as input a multi-layer graph $\mathcal{G}$ and two integers $d, s \in \mathbb{N}$. It removes the vertices v with Supp$(v) < s$ in iterations. In each iteration, we compute the d-core $C^d(G_i)$ on each layer i (lines 2–3) and then remove the vertices v from all layers in $\mathcal{G}$ if Supp$(v) < s$ (line 6). This process is repeated until Supp$(v) \geq s$ for all vertices v remaining in $\mathcal{G}$. Finally, the remaining graph $\mathcal{G}$ is returned as the result (line 8).

Limitations As verified by the experimental results in Sect. 8, 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 $\mathcal{F}$. Since $|\mathcal{F}| = \binom{l(\mathcal{G})}{s}$, we have $|\mathcal{F}| = \frac{(l(\mathcal{G})-s)^s}{\sqrt{2\pi s}s^s}(\frac{l(\mathcal{G})}{l(\mathcal{G})-s})^{l(\mathcal{G})+1/2}$ according to Stirling's approximation [9]. Obviously, for fixed s, $|\mathcal{F}|$ grows exponentially as $l(\mathcal{G})$ increases. When $\mathcal{F}$ cannot fit in main memory, we store all d-CCs on the disk, and the space cost is $O(\binom{l(\mathcal{G})}{s}n)$.
(2) The exponential growth on the size of $\mathcal{F}$ significantly increases the time cost for selecting k diversified d-CCs from $\mathcal{F}$ (lines 9–11). When $\mathcal{F}$ is stored on the disk, the I/O cost for d-CC selection is $O(\binom{l(\mathcal{G})}{s}nk/B)$, where B is the block size. The I/O cost is very high for large graphs.
(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.

5 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 the 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 [30], 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 Sect. 8, 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.

5.1 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 $|\mathcal{R}| < k$, we directly add C into $\mathcal{R}$ to enlarge its coverage.

Rule 2: For $C' \in \mathcal{R}$, let $\Delta(\mathcal{R}, C') = C' - \mathsf{Cov}(\mathcal{R} - \{C'\})$, $\Delta(\mathcal{R}, C')$ is the set of vertices in $\mathsf{Cov}(\mathcal{R})$ exclusively covered by C'. Let $C^*(\mathcal{R}) = \arg\min_{C' \in \mathcal{R}} |\Delta(\mathcal{R}, C')|$, $C^*(\mathcal{R})$ exclusively covers the least number of vertices by itself among all d-CCs in $\mathcal{R}$. If $|\mathcal{R}| = k$, we can replace $C^*(\mathcal{R})$ by a new d-CC C if the coverage of $\mathcal{R}$ can be enlarged by a sufficiently large factor. According to the framework for solving the *max-k-cover problem* [2], we replace $C^*(\mathcal{R})$ with C if $|\mathcal{R}| = k$ and

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

As proved in [2], applying these two rules can lead to a final result with guaranteed performance. The details of Update are described in Appendix B of this paper. By using two index structures, Update runs in $O(\max\{|C|, |C^*(\mathcal{R})|\})$ time.

5.2 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. 5, 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 $i \in L' - L$ satisfies $i > \max(L)$, 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. The depth-first search is realized by recursive Procedure BU-Gen described in Fig. 6. The BU-Gen takes as input a multi-layer graph $\mathcal{G}$, integers $d, s, k \in \mathbb{N}$, two-layer subset $L, L_Q \subseteq [l(\mathcal{G})]$, the d-CC $C^d_L(\mathcal{G})$ w.r.t. L, and the result set $\mathcal{R}$. The BU-Gen procedure works as follows.

Given the d-CC $C^d_L(\mathcal{G})$, we first expand L by adding a layer number j into L. Notably, the input layer subset L_Q records some layers that cannot be used to expand L. L_Q is generated by the pruning method which will be described later. Therefore, let $L_P = \{j | \max(L) < j \leq l(\mathcal{G})\} - L_Q$

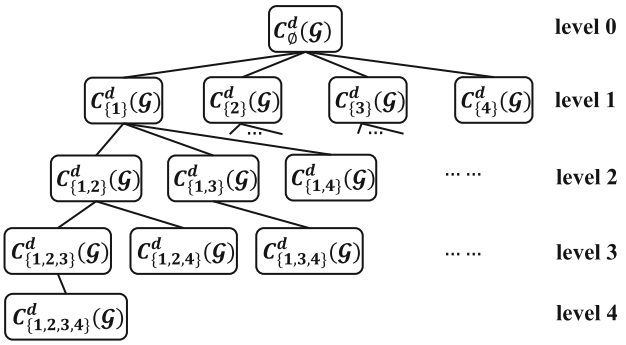


Fig. 5 Bottom-up search tree

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

Fig. 6 The BU-Gen procedure

(line 1). L_P is the set of layers potentially used to expand L. We also initialize a layer subset L_R to collect all layers in L_P that can be actually used to expand L (line 2). For each $j \in L_P$, 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 can 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 Sect. 3 (line 6 and line 20). Next, we process $C^d_{L'}(\mathcal{G})$ according to the following cases:

Case 1 (lines 7–8) If $|\mathcal{R}| < k$ and $|L'| = s$, we update $\mathcal{R}$ with $C^d_{L'}(\mathcal{G})$ by Rule 1 specified in Sect. 5.2.

Case 2 (lines 9–10) If $|\mathcal{R}| < k$ and $|L'| < s$, j can be used to expand L. Thus, we add j into L_R.

Case 3 (lines 21–22) If $|\mathcal{R}| = k$ and $|L'| = s$, we update $\mathcal{R}$ with $C^d_{L'}(\mathcal{G})$ by Rule 2 specified in Sect. 5.2.

Case 4 (lines 23–25) If $|\mathcal{R}| = k$ and $|L'| < s$, 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, j can be used to expand L so we add j into L_R. 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 of the bottom-up search, we exploit several pruning methods when $|\mathcal{R}| = k$ (Cases 3 and 4).

Method 1: support-based pruning (lines 12–13) For the d-CC $C^d_L(\mathcal{G})$, we need to add $s - |L|$ layers into L to obtain a d-CC on s layers to update $\mathcal{R}$. Let $D \subseteq L_P$ be a layer subset such that $|D| = s - |L|$, and let $I_D = \cap_{i \in D} C^d(G_i)$. Clearly, if $|\arg\max_D C^d_L(\mathcal{G}) \cap I_D| < \frac{1}{k}|\mathsf{Cov}(\mathcal{R})| + |\Delta(\mathcal{R}, C^*(\mathcal{R}))|$, none of the decedents of $C^d_L(\mathcal{G})$ can update $\mathcal{R}$. However, the computation of the exact D that maximizes $|C^d_L(\mathcal{G}) \cap I_D|$ is NP-Hard and hard to approximate [6]. Alternatively, we derive a bound for all I_D according to the following lemma.

Lemma 3 *Let I be the set of vertices that are contained in at least $s - |L|$ d-cores on the layers in L_P. For any subset $D \subseteq L_P$ such that $|D| = s - |L|$, we have $I_D \subseteq I$.*

With the bounding vertex subset I, we can stop searching the subtree rooted at $C^d_L(\mathcal{G})$ if we have $|C^d_L(\mathcal{G}) \cap I| < \frac{1}{k}|\mathsf{Cov}(\mathcal{R})| + |\Delta(\mathcal{R}, C^*(\mathcal{R}))|$. The correctness is ensured by the following lemma.

Lemma 4 (Support-based pruning) *For a d-CC $C^d_L(\mathcal{G})$ and the bounding vertex subset I, if we have $|C^d_L(\mathcal{G}) \cap I| < \frac{1}{k}|\mathsf{Cov}(\mathcal{R})| + |\Delta(\mathcal{R}, C^*(\mathcal{R}))|$, for any descendant $C^d_S(\mathcal{G})$ of $C^d_L(\mathcal{G})$ such that $|S| = s$, $C^d_S(\mathcal{G})$ cannot satisfy Eq. (1).*

Method 2: Order-based pruning (lines 14–17) For all $j \in L_P$, we can order the layer numbers j 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. 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 5 (Order-based pruning) *For a d-CC $C^d_L(\mathcal{G})$ and each $j \in L_P$, 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).*

Method 3: Internal d-CC computation pruning (line 20) Recall that in the dCC procedure, each time we remove an unpromising vertex that cannot exist in the d-CC from the graph. Therefore, the size of the input multi-layer graph decreases gradually. At some moment when the size of the remaining vertices is too small, the generated d-CC is impossible to update the result set $\mathcal{R}$. Therefore, we can immediately terminate the d-CC computation at this time. We call this pruning method the internal d-CC computation pruning.

We can apply this pruning method to the dCC procedure at line 20 of BU-Gen which computes $C^d_{L'}(\mathcal{G})$. To achieve this, several minor modifications need to be made to the dCC procedure. Specifically, the dCC procedure takes the size $\frac{1}{k}|\mathsf{Cov}(\mathcal{R})| + |\Delta(\mathcal{R}, C^*(\mathcal{R}))|$ as an additional input parameter. After line 18 of the dCC procedure, the unpromising vertex is removed from the graph, so we can check if the number of remaining vertices, i.e., $V(\mathcal{G})$, is less than $\frac{1}{k}|\mathsf{Cov}(\mathcal{R})| + |\Delta(\mathcal{R}, C^*(\mathcal{R}))|$. If so, the generated d-CC $C^d_{L'}(\mathcal{G})$ is unable to update $\mathcal{R}$. Therefore, we can immediately terminate the dCC procedure and safely skip the execution of lines 21–25 of the BU-Gen procedure. Otherwise, the dCC procedure works as usual. The correctness of this pruning method is obvious.

Method 4: Layer pruning (lines 27–29) For each $j \in L_P$, if $C^d_{L\cup\{j\}}(\mathcal{G})$ does not satisfy Eq. (1), we need not generate $C^d_S(\mathcal{G})$ for all S such that $L \cup \{j\} \subseteq S \subseteq [l(\mathcal{G})]$. The correctness is guaranteed by the following lemma.

Lemma 6 (Layer pruning) *For a d-CC $C^d_L(\mathcal{G})$ and each $j \in L_P$, if $C^d_{L\cup\{j\}}(\mathcal{G})$ does not satisfy Eq. (1), $C^d_{S\cup\{j\}}(\mathcal{G})$ cannot satisfy Eq. (1) for all S such that $L \subseteq S \subseteq [l(\mathcal{G})]$.*

For each $j \in L_R$, let $L' = L - \{j\}$ (line 28). By Lemma 6, any layer in $L_Q \cup (L_P - L_R)$ cannot be used to expand the d-CC $C^d_{L'}(\mathcal{G})$. As a result, for each L', we make a recursive call to BU-Gen with parameters $\mathcal{G}, d, s, k, L', L_Q \cup (L_P - L_R)$, $C^d_{L'}(\mathcal{G})$ and $\mathcal{R}$ (line 29).

5.3 The bottom-up algorithm

Figure 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($\mathcal{G}, d, s, k, \emptyset, \emptyset, V(\mathcal{G}), \mathcal{R}$) (line 4). To further speed up the algorithm, the preprocessing method (Procedure VertexDeletion) proposed in Sect. 4 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 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 $\mathcal{R}$ such that $|\mathcal{R}| = k$.

The InitTopK procedure takes as input a multi-layer graph $\mathcal{G}$, three integers $d, s, k \in \mathbb{N}$ and the set $\mathcal{R}$ of temporary top-k diversified d-CCs. First, we set $\mathcal{R}$ as an empty set

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

Procedure InitTopK$(\mathcal{G}, d, s, k, \mathcal{R})$
1: $\mathcal{R} \leftarrow \emptyset$
2: **for** $p \leftarrow 1$ to k **do**
3: $\quad i \leftarrow \arg\max_{i \in [l(\mathcal{G})]} |\mathsf{Cov}(\mathcal{R} \cup \{C^d(G_i)\})| - |\mathsf{Cov}(\mathcal{R})|$
4: $\quad L \leftarrow \{i\}$
5: $\quad C \leftarrow C^d(G_i)$
6: $\quad$ **for** $q \leftarrow 1$ to $s-1$ **do**
7: $\quad\quad j \leftarrow \arg\max_{j \in [l(\mathcal{G})]-L} |C \cap C^d(G_j)|$
8: $\quad\quad L \leftarrow L \cup \{j\}$
9: $\quad\quad C \leftarrow C \cap C^d(G_j)$
10: $\quad C' \leftarrow$ dCC$(\mathcal{G}[C], d, L)$
11: $\quad$ Update$(\mathcal{R}, C')$
12: **return** $\mathcal{R}$

Fig. 7 The BU-DCCS algorithm

(line 1). The **for** loop (lines 2–11) executes k times. In each loop, a candidate d-CC is added to $\mathcal{R}$ in the following way: First, we select layer i such that the d-core $C^d(G_i)$ can maximally enlarges $\mathsf{Cov}(\mathcal{R})$ (line 3). Let $C = C^d(G_i)$ and $L = \{i\}$ (lines 4–5). Then, we add $s-1$ other layer numbers to L in a greedy manner. In each time, we choose layer $j \in [l(\mathcal{G})] - L$ that maximizes $|C \cap C^d(G_j)|$, update L to $L \cup \{j\}$, and update C to $C \cap C^d(G_j)$ (lines 7–9). When $|L| = s$, we compute the d-CC $C^d_L(\mathcal{G})$ and update $\mathcal{R}$ with $C^d_L(\mathcal{G})$ (lines 11–12).

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

6 The top-down algorithm

The bottom-up algorithm traverses a search tree from the root down to level s. When $s \geq l(\mathcal{G})/2$, the efficiency of the algorithm degrades significantly. As verified by the experiments in Sect. 8, the performance of the bottom-up algorithm is close to or even worse than the greedy algorithm when $s \geq l(\mathcal{G})/2$. To handle this issue, we propose a top-down approach for the DCCS problem when $s \geq l(\mathcal{G})/2$.

In this section, we assume $s \geq l(\mathcal{G})/2$. 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. Given that we now have a d-CC C w.r.t. layer subset L, we generate the d-CC C' w.r.t. layer subset L' such that $L' \subseteq L$ in the top-down algorithm. Obviously, we have $C \subseteq C'$, so the pruning techniques in the bottom-up algorithm based on the containment property (Property 3) of d-CC are certainly not applicable. Therefore, we must propose a series of new pruning techniques suitable for the top-down search. Specifically, for each d-CC, we associate it with a potential set that contains all vertices in the descendants of this d-CC in the top-down search tree. We observe that the potential set satisfies the containment property. Let U and U' be the potential set of C and C', respectively. We have $C' \subseteq U'$ and $U' \subseteq U$. Therefore, if U' is unlikely to improve the quality of the result, none of the descendants of C' can do. The top-down algorithm also has an approximation ratio of 1/4. As verified by the experiments in Sect. 8, the top-down algorithm is superior to the other algorithms when $s \geq l(\mathcal{G})/2$.

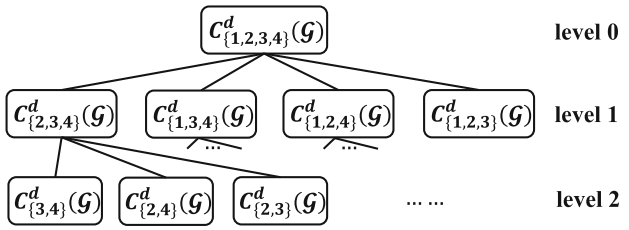


Fig. 8 Top-down search tree

6.1 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. 8, 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 $i \in L - L'$ satisfies $i > \max([l(\mathcal{G})] - L)$. Except the root $C^d_{[l(\mathcal{G})]}$, all d-CCs in the search tree have 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 6.2 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. 9. The arrows in Fig. 9 indicate 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 6.2 will describe how to compute $U^d_{L'}(\mathcal{G})$ in details.

The top-down candidate d-CC generation is implemented by the recursive procedure TD-Gen in Fig. 10. Let $L_R = \{j \,|\, \max([l(\mathcal{G})] - L) < j \leq l(\mathcal{G})\} \cap L$ 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 $U^d_{L'}(\mathcal{G})$ by the method in Sect. 6.2

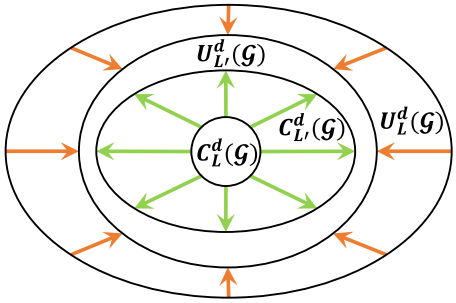


Fig. 9 Relationships between $C^d_L(\mathcal{G})$, $U^d_L(\mathcal{G})$, $C^d_{L'}(\mathcal{G})$, and $U^d_{L'}(\mathcal{G})$

Procedure TD-Gen($\mathcal{G}, d, s, k, L, C^d_L(\mathcal{G}), U^d_L(\mathcal{G}), \mathcal{R}$)

1: $L_R = \{j \mid \max([l(\mathcal{G})] - L) < j \leq l(\mathcal{G})\} \cap L$
2: **for** each $j \in L_R$ **do**
3: $L' \leftarrow L - \{j\}$
4: $U^d_{L'}(\mathcal{G}) \leftarrow$ RefineU($\mathcal{G}, d, s, L', U^d_L(\mathcal{G})$)
5: **if** $|\mathcal{R}| < k$ **then**
6: **for** each $j \in L_R$ **do**
7: $L' \leftarrow L - \{j\}$
8: $C^d_{L'}(\mathcal{G}) \leftarrow$ dCC($\mathcal{G}[U^d_{L'}(\mathcal{G})], d, L'$)
9: **if** $|L'| = s$ **then**
10: Update($\mathcal{R}, C^d_{L'}(\mathcal{G})$)
11: **else**
12: TD-Gen($\mathcal{G}, d, s, k, L, C^d_{L'}(\mathcal{G}), U^d_{L'}(\mathcal{G}), \mathcal{R}$)
13: **else**
14: sort $j \in L_R$ in descending order of $|U^d_{L-\{j\}}(\mathcal{G})|$
15: **for** each j in the sorted L_R **do**
16: $L' \leftarrow L - \{j\}$
17: **if** $|U^d_{L'}(\mathcal{G})| < \frac{1}{k}|\mathsf{Cov}(\mathcal{R})| + |\Delta(\mathcal{R}, C^*(\mathcal{R}))|$ **then**
18: **break**
19: **else**
20: $C^d_{L'}(\mathcal{G}) \leftarrow$ dCC($\mathcal{G}[U^d_{L'}(\mathcal{G})], d, L'$)
21: **if** $|L'| = s$ **then**
22: Update($\mathcal{R}, C^d_{L'}(\mathcal{G})$)
23: **else**
24: **if** $U^d_{L'}(\mathcal{G})$ satisfies Eq. (1) **then**
25: **if** $U^d_{L'}(\mathcal{G})$ satisfies Eq. (2) **then**
26: $S \leftarrow L' - \{|L'| - s$ numbers randomly chosen from $L_R\}$
27: $C^d_S(\mathcal{G}) \leftarrow$ dCC($\mathcal{G}[U^d_{L'}(\mathcal{G})], d, S$)
28: Update($\mathcal{R}, C^d_S(\mathcal{G})$)
29: **else**
30: TD-Gen($\mathcal{G}, d, s, k, L, C^d_{L'}(\mathcal{G}), U^d_{L'}(\mathcal{G}), \mathcal{R}$)

Fig. 10 The TD-Gen procedure

(line 4). After obtaining $U^d_{L'}(\mathcal{G})$, $C^d_{L'}(\mathcal{G})$ can be easily computed by applying the dCC procedure on input $\mathcal{G}[U^d_{L'}(\mathcal{G})]$, d and L' (lines 8 and 20). 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 Sect. 5.2.

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 21–22) If $|\mathcal{R}| = k$ and $|L'| = s$, we update $\mathcal{R}$ with $C^d_{L'}(\mathcal{G})$ by Rule 2 specified in Sect. 5.2.

Case 4 (lines 23–30) Similar to Lemma 2, if $|\mathcal{R}| = k$ and $|L'| > s$, we apply $U^d_{L'}(\mathcal{G})$ to check whether to extend the descendants of $C^d_{L'}(\mathcal{G})$ (line 24).

Lemma 7 (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), we present several pruning methods to further speed up the top-down search process as follows.

Method 1: Order-based pruning (lines 14–17) Similar to Lemma 5, we can also order the layer numbers $j \in L_R$ in descending order of $|U^d_{L-\{j\}}(\mathcal{G})|$ (line 14) and prune some subtrees earlier (lines 17–18).

Lemma 8 (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 $|U^d_{L-\{j\}}(\mathcal{G})| < \frac{1}{k}|\mathsf{Cov}(\mathcal{R})| + |\Delta(\mathcal{R}, C^*(\mathcal{R}))|$, any descendant $C^d_{L-\{j\}}(\mathcal{G})$ of $C^d_L(\mathcal{G})$ cannot satisfy Eq. (1).*

Method 2: Potential set pruning (lines 25–28) 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 26–28). The correctness is ensured by the following lemma.

Lemma 9 (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*

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

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.

6.2 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 $i \in L$ and $i < \max([l(\mathcal{G})] - L)$, layer i 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 $i \in S$.

Class 2: By the relationship of d-CCs in the top-down search tree, for any layer number $i \in L$ and $i > \max([l(\mathcal{G})] - L)$, layer i 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 $i \in S$.

Procedure RefineU$(\mathcal{G}, d, s, L', U_L^d(\mathcal{G}))$
1: $U \leftarrow U_L^d(\mathcal{G})$
2: $M_{L'} \leftarrow \{j | j \in L, j < \max([l(\mathcal{G})] - L)\}$; $N_{L'} \leftarrow L - M_{L'}$
3: **repeat**
4: **while** there exists $v \in U$ and $i \in M_{L'}$ such that $d_{G_i[U]}(v) < d$ **do**
5: remove v from U and from all layers of $\mathcal{G}$
6: **while** there exists $v \in U$ that occurs in less than $s - |M_{L'}|$ of the d-cores $C^d(G_j)$ for $j \in N_{L'}$ **do**
7: remove v from U and from all layers of $\mathcal{G}$
8: **until** no vertex in U can be removed
9: **return** U

Fig. 11 The RefineU procedure

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: $C^d_{[l(\mathcal{G})]} \leftarrow$ dCC$(\mathcal{G}, d, [l(\mathcal{G})])$
5: TD-Gen$(\mathcal{G}, d, s, k, [l(\mathcal{G})], C^d_{[l(\mathcal{G})]}, V(\mathcal{G}), \mathcal{R})$
6: **return** $\mathcal{R}$

Fig. 12 The TD-DCCS algorithm

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. 11 refines $U_L^d(\mathcal{G})$ to $U_{L'}^d(\mathcal{G})$. Let $U = U_L^d(\mathcal{G})$ (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 $U_{L'}^d(\mathcal{G})$ (line 9).

Refinement Method 1 (lines 4–5) For each layer number $i \in M_{L'}$, we have $i \in S$ for all descendants $C_S^d(\mathcal{G})$ of $C_{L'}^d(\mathcal{G})$ with $|S| = s$. Note that $C_S^d(\mathcal{G})$ 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 $v \notin C_S^d(\mathcal{G})$, 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 $C_S^d(\mathcal{G})$ of $C_L^d(\mathcal{G})$ 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 $\mathcal{G}$.

6.3 Top-down algorithm

We present the complete top-down DCCS algorithm TD-DCCS in Fig. 12. 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. Next, we compute $C^d_{[l(\mathcal{G})]}$ (line 4) and invoke recursive Procedure TD-Gen$(\mathcal{G}, d, s, k, [l(\mathcal{G})], C^d_{[l(\mathcal{G})]}, V(\mathcal{G}), \mathcal{R})$ to generate candidate d-CCs and update the result set $\mathcal{R}$ (line 5). Finally, $\mathcal{R}$ is returned as the result (line 6).

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

7 Optimized algorithms

In this section, we propose several optimized algorithms for fast finding the diversified d-CCs in a multi-layer graph. In all aforementioned algorithms for the DCCS problem, we invoke the dCC procedure proposed in Sect. 3 to compute d-CCs in the search process. However, this method is still not efficient enough since it involves lots of redundant examinations of vertices. In this section, we introduce an index structure, which organizes all vertices of the input multi-layer graph hierarchically. Base on this index, we propose a faster d-CC computation method with less examination of vertices. By applying this method to the previous DCCS algorithms, the experimental results in Sect. 8 show that all of the optimized algorithms run faster than the original algorithms.

7.1 The index structure

We introduce the index structure in this subsection. The index I organizes all the vertices of $\mathcal{G}$ hierarchically and helps filter out the vertices irrelevant to $C_{L'}^d(\mathcal{G})$ efficiently. Recall that Supp(v) is the number of layers whose d-cores contain v. The index I is constructed based on Supp(v). Specifically, for $1 \leq h \leq l(\mathcal{G})$, let J_h be the set of vertices v iteratively removed from $\mathcal{G}$ due to Supp$(v) \leq h$. 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 $\mathcal{G}$ [1].

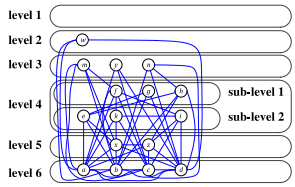
We present the BuildIndex procedure for constructing the index in Fig. 13. BuildIndex takes a multi-layer graph $\mathcal{G}$ and an integer $d \in \mathbb{N}$ as input. At the very beginning, we compute the d-core on each layer (line 2) and initialize the index I as empty (line 3). The index I is basically the hierarchy of the vertices following $I_1, I_2, \ldots, I_{l(\mathcal{G})}$, that is, the vertices in I_{i+1} are placed on higher levels than those in I_i. Internally, the vertices in I_i are also placed on a stack of levels. Initially, we set $h = 1$ (line 4). For each h, vertices in I_h are placed 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 Supp$(v) \leq i$, they are actually removed in different batches. In each batch (lines 7–12), we select all the vertices v with Supp$(v) \leq i$ into S (line 7) and remove them together from all layers of $\mathcal{G}$ (line 12). 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 (line 10). We associate each vertex v in the index with $L(v)$ (line 11).

[1] We do not need to consider I_0 since vertices in I_0 are not in the d-core on any layer of the multi-layer graph $\mathcal{G}$.

Procedure BuildIndex($\mathcal{G}, d$)
1: **for** $i \leftarrow 1$ **to** $l(\mathcal{G})$ **do**
2: compute the d-core $C^d(G_i)$ on graph G_i
3: initialize an empty index I
4: $h \leftarrow 1$
5: **while** $V(\mathcal{G}) \neq \emptyset$ **do**
6: **repeat**
7: $S \leftarrow \{v | v \in V(\mathcal{G}), \mathsf{Supp}(v) \leq h\}$
8: add S as a new level in the index I
9: **for** each vertex $v \in S$ **do**
10: $L(v) \leftarrow \{i | v \in C^d(G_i)\}$
11: attach $L(v)$ to vertex v
12: remove v from all layers of $\mathcal{G}$
13: **for** $i \leftarrow 1$ **to** $l(\mathcal{G})$ **do**
14: update the d-core $C^d(G_i)$ on graph G_i
15: **until** $S = \emptyset$
16: $h \leftarrow h + 1$
17: **for** each pair of vertices u, v in the index I **do**
18: **if** (u, v) is an edge on a layer of the original graph $\mathcal{G}$ **then**
19: add an edge between u and v in I
20: **return** I

Fig. 13 The BuildIndex procedure



Vertex	Layer Numbers $L(v)$	Vertex	Layer Numbers $L(v)$
w	$\{1,6\}$	k	$\{6\}$
m	$\{1,3,6\}$	i	$\{5,6\}$
y	$\{1,5,6\}$	k	$\{1,2,3,4,5\}$
n	$\{1,4,6\}$	z	$\{2,3,4,5,6\}$
f	$\{1,3,4,5\}$	a	$\{1,2,3,4,5,6\}$
g	$\{2,3,4,6\}$	b	$\{1,2,3,4,5,6\}$
h	$\{1,2,5,6\}$	c	$\{1,2,3,4,5,6\}$
e	$\emptyset$	d	$\{1,2,3,4,5,6\}$

Fig. 14 Illustration of index structure

After a batch, due to the removal of vertices, we need to update the d-core on each layer (line 14). After that, some vertices v originally satisfying $\mathsf{Supp}(v) > i$ may become $\mathsf{Supp}(v) \leq i$ and thus will be removed in next batch. Therefore, in I_i, the vertices removed in the same batch are place on the same sub-level, and the vertices removed in a later batch are placed on a higher sub-level than the vertices removed in an early batch. We repeat the removal process until S is empty. After that, we increase h by 1 (line 16) and continue to process vertices in I_{h+1}. The iteration is repeated until all vertices have been removed from the graph.

After placing all vertices of $\mathcal{G}$ into the index I, we add an edge between vertices u and v in the index if (u, v) is an edge on a layer of $\mathcal{G}$ (line 19). Finally, the index I is returned as the result (line 20).

Complexity analysis Let $n = |V(\mathcal{G})|$, $l = l(\mathcal{G})$, $m_i = |E_i(\mathcal{G})|$, $m = |\bigcup_{i \in [l(\mathcal{G})]} E_i(\mathcal{G})|$, and $m' = \sum_{i \in [l(\mathcal{G})]} m_i$. By [3], the d-core computation consumes $O(\sum_{i \in [l(\mathcal{G})]} m_i) = O(m')$ time. Let Δ be the maximum degree of a vertex across all layers. Note that, for each $1 \leq h \leq l(\mathcal{G})$, there exist at most Δ batches. This is because if the support number of a vertex decreases, at least one of its neighbors is removed from the graph. Thus, the time complexity of the index building process is at most $O(m' + \Delta m l)$. Obviously, the space to store the index is at most $O(nl + m)$.

For ease of understanding, we illustrate an example of the index structure in Fig. 14 for the multi-layer graph $\mathcal{G}$ shown in Fig. 2 when $d = 3$. In the index, level h represents all vertices in I_h. In general, the vertices in I_h may be removed in a sequence of batches. The vertices removed together in a batch forms a sub-level of level h.

Initially ($h = 1$), we have $\mathsf{Supp}(v) > 1$ for all vertices v in $\mathcal{G}$, so level 1 contains no vertex. Then, h is increased to 2, and we have $\mathsf{Supp}(w) = 2$ because $w \in C^d(G_1)$ and $w \in C^d(G_6)$. For each vertex v in $\mathcal{G}$, the table in Fig. 14 lists $L(v)$, the set of layer numbers on which v is contained in the d-cores just before v is removed from $\mathcal{G}$. Hence, we place w on level 2 and remove it from $\mathcal{G}$. After that, we have $\mathsf{Supp}(v) > 2$ for all vertices v remaining in $\mathcal{G}$, so we set $h = 3$. Now, we have $\mathsf{Supp}(m) = \mathsf{Supp}(y) = \mathsf{Supp}(n) = 3$, so they are placed on level 3 and removed from $\mathcal{G}$. Since no vertex v in $\mathcal{G}$ now satisfies $\mathsf{Supp}(v) > 3$, we increase h to 4. At this time, we have $\mathsf{Supp}(f) = \mathsf{Supp}(g) = \mathsf{Supp}(h) = 4$ and $\mathsf{Supp}(e) = \mathsf{Supp}(k) = \mathsf{Supp}(i) = 5$. Thus, vertices f, g, and h are removed together in the same batch, so f, g, and h are placed on the first sub-level of level 4. After removing f, g, and h, we have $\mathsf{Supp}(e) = 0$, $\mathsf{Supp}(k) = 1$, and $\mathsf{Supp}(i) = 2$. Hence, vertices e, k, and i must be removed in the same batch, so they are placed on the second sub-level of level 4. After removing e, k, and i, no vertex v in $\mathcal{G}$ satisfies $\mathsf{Supp}(v) \leq 4$, so we increase h to 5. Later, vertices x and z are placed on level 5. Finally, h is increased to 6, and vertices a, b, c, and d are placed on level 6.

7.2 The faster *d*-CC computation method

In this subsection, we propose a faster d-CC computation method based on the index structure.

Main idea Recall that, in the dCC procedure proposed in Sect. 3, we need to repeatedly check whether each remaining vertex satisfies the degree constraint of the d-CC. Clearly, there involve lots of redundant vertex examinations. To alleviate this, in the fast d-CC computation method, we accelerate the d-CC computation process by using the following two strategies.

Strategy 1: Safely eliminating vertices without examination In the dCC procedure, for each remaining vertex v, we need to check the degree of v on each layer to decide whether to remove v. However, if we have already known that v must not exist in the d-CC by other means in advance, we can directly eliminate it without examination.

Strategy 2: Terminating vertices' examination early During the dCC procedure, if a vertex v is removed from the graph, for each vertex u adjacent to v on some layer, we need to update the degree of u on each layer, whereas, if we have already known that u must in or must not in the d-CC, there is no need to update u's degree afterward. In other words, we can terminate the examination of such vertices early.

The two strategies can help reduce lots of redundant vertex examinations in the d-CC computation. In the following, we describe their implementation details.

Details of Strategy 1 By exploiting the index structure, we can detect some vertices that must not in the d-CC $C_L^d(\mathcal{G})$ before examination. To this end, we first introduce some useful concepts as follows.

For each vertex w in the index, we call w a *candidate vertex* if $L \subseteq L(w)$. w is possible to exist in $C_L^d(\mathcal{G})$. For any two vertices w and z in the index, we denote $w \prec z$ if z is placed on a higher level than w and (w, z) is an edge in the index. If there exists a sequence of vertices $w_0, w_1, \ldots, w_n$ in the index such that w_0 is a candidate vertex, and $w_i \prec w_{i+1}$ for $0 \leq i < n$, we say there exists a *candidate path* in the index to the vertex w_n.

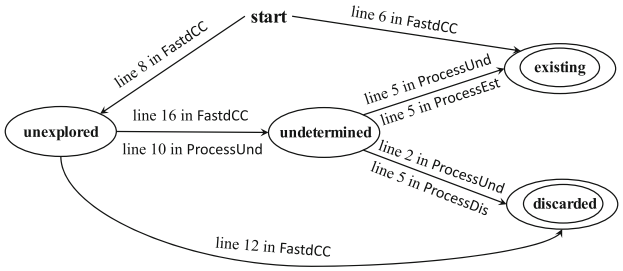
With the concept of candidate path, the following lemma states a necessary condition for a vertex w existing in $C_L^d(\mathcal{G})$. In the d-CC computation, we can directly eliminate all vertices that do not satisfy the following condition.

Lemma 10 *For each vertex $w \in C_L^d(\mathcal{G})$, there must exist a candidate path in the index to w.*

For example, consider the index in Fig. 14. Let $L = \{3, 4, 5\}$. We find that vertex w does not satisfy $L \subseteq L(w)$, so w is not in $C_L^d(\mathcal{G})$. In the next level, there are no candidate paths to vertices m, y, and n, so they can also be removed. On the remaining levels, we can also remove vertices g, h and e. Therefore, we only need to examine vertices a, b, c, d, f, k, i, x, and z.

Details of Strategy 2 To achieve the early termination, we set each vertex v in the index to be in one of the following four states:

- *discarded* if it has been determined that $v \notin C_{L'}^d(\mathcal{G})$;
- *existing* if it has been determined that $v \in C_{L'}^d(\mathcal{G})$;
- *undetermined* if v has been checked but has not be determined if $v \in C_{L'}^d(\mathcal{G})$;
- *unexplored* if it has not been checked by the search process.



In the d-CC computation process, a discarded vertex or an existing vertex will not be involved in the following computation; an undetermined vertex may become discarded due to the deletion of some edges or become existing if it connects to sufficient number of existing vertices; an unexplored vertex will become undetermined after examination or directly become discarded if it does not satisfy the condition stated in Lemma 10.

Procedure ProcessUnd$(\mathcal{G}, v, d, L, I)$
1: **if** $d_i^+(v) < d$ for some $i \in L$ **then**
2: $\texttt{state}(v) \leftarrow$ discarded
3: ProcessDis$(\mathcal{G}, v, d, L, I)$
4: **else if** $d_i^*(v) \geq d$ for each $i \in L$ **then**
5: $\texttt{state}(v) \leftarrow$ existing
6: ProcessEst$(\mathcal{G}, v, d, L, I)$
7: **else**
8: **for** each vertex u such that $v \prec u$ **do**
9: **if** $\texttt{state}(u) =$ unexplored **then**
10: $\texttt{state}(u) \leftarrow$ undetermined
11: ProcessUnd$(\mathcal{G}, u, d, L, I)$

Procedure ProcessDis$(\mathcal{G}, v, d, L, I)$
1: **for** each vertex u adjacent to v in the index I **do**
2: **if** $\texttt{state}(v) =$ undetermined or unexplored **then**
3: $d_i^+(u) \leftarrow d_i^+(u) - 1$ for each $i \in L$ and $(u, v) \in E_i(\mathcal{G})$
4: **if** $d_i^+(u) < d$ for some $i \in L$ **then**
5: $\texttt{state}(u) \leftarrow$ discarded
6: ProcessDis$(\mathcal{G}, u, d, L, I)$

Procedure ProcessEst$(\mathcal{G}, v, d, L, I)$
1: **for** each vertex u adjacent to v in the index I **do**
2: **if** $\texttt{state}(v) =$ undetermined or unexplored **then**
3: $d_i^*(u) \leftarrow d_i^*(u) + 1$ for each $i \in L$ and $(u, v) \in E_i(\mathcal{G})$
4: **if** $d_i^*(u) \geq d$ for each $i \in L$ **then**
5: $\texttt{state}(u) \leftarrow$ existing
6: ProcessEst$(\mathcal{G}, u, d, L, I)$

Fig. 15 The ProcessUnd, ProcessDis, and ProcessEst procedure

For each vertex v whose state is firstly setting to be undetermined, discarded, and existing, we use the ProcessUnd procedure, the ProcessDis procedure and the ProcessEst procedure (in Fig. 15) to process the effects to other vertices of changing v's state, respectively. The details of the three procedures are elaborated as follows.

Procedure ProcessUnd For each $i \in L$, let $d_i^+(v)$ be the number of non-discarded vertices adjacent to v in G_i and let $d_i^*(v)$ be the number of existing vertices adjacent to v in G_i. Clearly, $d_i^+(v)$ is an upper bound on v's degree in G_i. If $d_i^+(v) < d$ for some $i \in L$, we must have $v \notin C_L^d(\mathcal{G})$, so we set v as discarded (line 2) and invoke the ProcessDis procedure on v (line 3). If $d_i^*(v) \geq d$ for each $i \in L$, we must have $v \in C_L^d(\mathcal{G})$, so we set v as existing (line 5) and invoke the ProcessEst procedure on v (line 6). Otherwise, since v is undetermined, there may exist a candidate path to each vertex u such that $v \prec u$. By Lemma 10, u may exist in $C_L^d(\mathcal{G})$. Therefore, if u is unexplored, we set u as undetermined (line 10) and recursively invoke the ProcessUnd procedure to further check the vertex u (line 11).

Procedure ProcessDis If v is set as discarded, the removal of v may trigger the removal of other vertices. Therefore, for each undetermined or unexplored vertex u adjacent to v in the index, we decrease $d_i^+(u)$ by 1 if (u, v) is an edge on a layer $i \in L$ (line 3). If $d_i^+(u) < d$ for some $i \in L$, we also set u as discarded (line 5) and recursively invoke the ProcessDis procedure on u (line 6).

Procedure ProcessEst If v is set as existing, we may find more existing vertices from v. Specifically, for each undetermined or unexplored vertex u adjacent to v in the index I, we increase $d_i^*(u)$ by 1 if (u, v) is an edge on a layer $i \in L$

Procedure FastdCC($\mathcal{G}, d, s, L, I, X, Y$)
1: $Z = Y \cap (\bigcup_{h=|L|}^{l(\mathcal{G})} I_h)$
2: removed all vertices not in Z from the graph $\mathcal{G}$ and the index I
3: **for** each vertex $v \in Z$ **do**
4: compute $d_i^+(v)$ and $d_i^*(v)$ of all layer $i \in L$
5: **if** $v \in X$ **then**
6: state(v) $\leftarrow$ existing
7: **else**
8: state(v) $\leftarrow$ unexplored
9: **for** each level of the index **do**
10: **if** there exist some undetermined vertices on the level **then**
11: **for** each unexplored vertex v on the level **do**
12: state(v) $\leftarrow$ discarded
13: ProcessDis($\mathcal{G}, v, d, L, I$)
14: **else**
15: **for** each unexplored vertex v on the level **do**
16: state(v) $\leftarrow$ undetermined
17: ProcessUnd($\mathcal{G}, v, d, L, I$)
18: **return** $C_L^d(\mathcal{G}) \leftarrow \{v | \text{state}(v) = \text{undetermined or existing}\}$

Fig. 16 The FastdCC procedure

(line 3). If $d_i^*(u) \geq d$ for each $i \in L$, we also set u as existing (line 5) and recursively invoke the ProcessEst procedure on u (line 6).

Fast d-CC computation We present the FastdCC procedure to faster compute the d-CC in Fig. 16. The input of FastdCC includes the multi-layer graph $\mathcal{G}$, integers $d, s \in \mathbb{N}$, the layer subset L, the index I, and two vertices subset X and Y. The two vertices subset X and Y satisfy that $X \subseteq C_L^d(\mathcal{G}) \subseteq Y$. At the very beginning, we obtain the vertex subset $Z = Y \cap (\bigcup_{h=|L'|}^{l(\mathcal{G})} I_h)$ (line 1) and remove all vertices not in Z from the graph $\mathcal{G}$ and the index I (line 2). This is because we only need to consider vertices in Z to compute $C_L^d(\mathcal{G})$ by the following lemma.

Lemma 11 $C_L^d(\mathcal{G}) \subseteq Y \cap \left(\bigcup_{h=|L|}^{l(\mathcal{G})} I_h\right)$.

Before the search starts, we compute $d_i^+(v)$ and $d_i^*(v)$ for all vertices $v \in V(\mathcal{G})$ and for all layers $i \in L$ (line 4). Initially, the state of a vertex v of $\mathcal{G}$ is set to be existing if it is in X (line 6) and unexplored otherwise (line 8).

In the main search process, we check the vertices in the index in a level-by-level fashion from lower levels to higher levels. In each iteration (lines 10–17), we examine all vertices on a level of the index. Based on the states of vertices on the level, they are processed in two cases:

Case 1 (lines 10–13) There exist some undetermined vertices in the level. At this time, since all vertices in lower levels have been examined, it implies that there exist no candidate paths to each unexplored vertex v in the level. By Lemma 10, we must have $v \notin C_L^d(\mathcal{G})$. Therefore, we can directly set v to be discarded (line 12) and invoke Procedure ProcessDis on v (line 13).

Case 2 (lines 14–17) Otherwise, each unexplored vertex v on the level is potential to exist in $C_L^d(\mathcal{G})$. Therefore, we set v as undetermined (line 16) and invoke Procedure ProcessUnd to further check v (line 17).

After examining all levels in the index, $C_L^d(\mathcal{G})$ is exactly the set of all undetermined and existing vertices (line 18).

Following the previous example, let $L = \{3, 4, 5\}$, $X = \{a, b, c, d\}$, and $Y = \{a, b, c, d, f, k, i, x, z\}$. All vertices in X are set to be existing. First, we examine vertex f in the lowest level. f remains to be undetermined. Therefore, we further check vertices k and i in the next level of f. For vertex k, we have $d_3^+(k) = 1$ and $d_3^-(k) = 0$, so k is set to be discarded. Then, we proceed to check k's neighbors. For vertex i, we have $d_4^+(i) = 1$ and $d_4^-(i) = 1$, so i is also set to be discarded. Next, we have $d_3^+(f) = 1$ and $d_3^-(f) = 1$. Therefore, f is set to be discarded. After that, we check vertices x and z. The states of x and z remain undetermined. Finally, we obtain $C_{\{3,4,5\}}^d = \{a, b, c, d, x, z\}$.

Correctness analysis The correctness of the FastdCC procedure can be guaranteed by the following lemma.

Lemma 12 *For any vertex $v \notin C_L^d(\mathcal{G})$, v must be set to discarded in the FastdCC procedure.*

Complexity analysis Let $n = |V(\mathcal{G})|$, $l = l(\mathcal{G})$, $m_i = |E_i(\mathcal{G})|$, $m = |\bigcup_{i \in [l(\mathcal{G})]} E_i(\mathcal{G})|$, and $m' = \sum_{i \in [l(\mathcal{G})]} m_i$. The following lemma states the time complexity of FastdCC. Since $m' \leq ml$, the time complexity of FastdCC is always lower than that of dCC. For the space complexity, the FastdCC procedure needs $O(n)$ extra space to store the states of all vertices and $O(nl)$ extra space to $d_i^+(v)$ and $d_i^*(v)$ of all vertices in all layers in L. Therefore, the space complexity of FastdCC is $O(n + nl) = O(nl)$.

Lemma 13 *The time complexity of the FastdCC procedure is $O(nl + m')$.*

7.3 The optimized algorithms

We present several optimized DCCS algorithms, namely GD-DCCS+, BU-DCCS+, TD-DCCS+, in this subsection.

Optimized greedy algorithm The GD-DCCS+ algorithm can be simply obtained by two minor modifications of the GD-DCCS algorithm:

(1) Before line 2 of GD-DCCS, we add a statement "$I \leftarrow$ BuildIndex($\mathcal{G}, d$)" to construct the index I.
(2) Line 7 of GD-DCCS is replaced by the statement "$C_L^d(\mathcal{G}) \leftarrow$ FastdCC($\mathcal{G}, d, s, L, I, \emptyset, S$)."

Optimized bottom-up algorithm The BU-DCCS+ algorithm can be simply obtained by several minor modifications of the BU-DCCS algorithm, the InitTopk procedure, and the BU-Gen procedure:

(1) Before line 4 of BU-DCCS, we add a statement "$I \leftarrow$ BuildIndex($\mathcal{G}, d$)" to construct the index I.
(2) InitTopK adds the index I as an input parameter. Line 10 of InitTopK is replaced by the statement "$C' \leftarrow$ FastdCC($\mathcal{G}, d, s, L, I, \emptyset, C$)."

(3) BU-Gen adds the index I as an input parameter. Line 6 and line 20 of BU-Gen are replaced by the statement "$C^d_{L'}(\mathcal{G}) \leftarrow$ FastdCC$(\mathcal{G}, d, s, L', I, \emptyset, C^d_L(\mathcal{G}) \cap C^d(G_j))$."
(4) The internal d-CC computation pruning method proposed in Sect. 5.2 needs to be adapted to be applied in the BU-Gen procedure. In the FastdCC procedure, we can also terminate the computation of d-CC early sometimes. Specifically, the FastdCC procedure, and the ProcessDis procedure take the size $\frac{1}{k}|\mathsf{Cov}(\mathcal{R})| + |\Delta(\mathcal{R}, C^*(\mathcal{R}))|$ as an input parameter. In the ProcessDis procedure, we check whether the number of the non-discarded vertices is less than $\frac{1}{k}|\mathsf{Cov}(\mathcal{R})| + |\Delta(\mathcal{R}, C^*(\mathcal{R}))|$ at the very beginning. If so, we can immediately terminate the FastdCC procedure and safely skip the execution of lines 21–25 in the BU-Gen procedure.

Optimized top-down algorithm The TD-DCCS+ algorithm can be simply obtained by several minor modifications of the TD-DCCS algorithm, the InitTopk procedure, and the TD-Gen procedure:

(1) Before line 4 of TD-DCCS, we add a statement "$I \leftarrow$ BuildIndex$(\mathcal{G}, d)$" to construct the index I.
(2) Line 4 of TD-DCCS is replaced by the statement "$C^d_{[l(\mathcal{G})]}(\mathcal{G}) \leftarrow$ FastdCC$(\mathcal{G}, d, s, L, I, \emptyset, V(\mathcal{G}))$."
(3) InitTopK adds the index I as an input parameter. Line 10 of InitTopK is replaced by the statement "$C' \leftarrow$ FastdCC$(\mathcal{G}, d, s, L, I, \emptyset, C)$."
(4) TD-Gen adds the index I as an input parameter. Line 8 and line 20 of TD-Gen are replaced by the statement "$C^d_{L'}(\mathcal{G}) \leftarrow$ FastdCC$(\mathcal{G}, d, s, L', I, C^d_L(\mathcal{G}), U^d_{L'}(\mathcal{G}))$." Line 27 of TD-Gen is replaced by the statement "$C^d_S(\mathcal{G}) \leftarrow$ FastdCC$(\mathcal{G}, d, s, L', I, \emptyset, U^d_{L'}(\mathcal{G}))$."

Notably, the algorithms GD-DCCS$^+$, BU-DCCS$^+$, and TD-DCCS$^+$ have the same approximation ratios of GD-DCCS, BU-DCCS, and TD-DCCS, respectively.

8 Performance evaluation

8.1 Experimental setting

Algorithms We implemented all of the proposed algorithms in this paper, including GD-DCCS, BU-DCCS, TD-DCCS, GD-DCCS+, BU-DCCS+, and TD-DCCS+, in C++ and experimentally evaluated them in this section. We designate GD-DCCS as the baseline. Each algorithm is evaluated by its execution time (efficiency) and the result 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. 17 Statistics of graph datasets used in experiments

Datasets We use six real-world graph datasets of various types and sizes in the experiments. The statistics of the graph datasets are summarized in Fig. 17. *PPI* is a protein–protein interaction network extracted from the STRING DB[2]. It contains eight layers representing the interactions between proteins detected by different methods. The dataset *Author* is a co-authorship network obtained from AMiner[3]. It contains ten layers representing the collaboration between authors in ten different years. The other datasets were obtained from the KONECT[4] and SNAP [5], 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 five 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. 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. 18. Notably, 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.

8.2 Experimental results

Execution time w.r.t. parameter k We evaluate the execution time of all of the algorithms w.r.t. parameter k. At first, we experiment for small s. Since the TD-DCCS and TD-DCCS+ algorithms are not applicable when $s < l(\mathcal{G})/2$, we only test the other four algorithms for small s. Figure 19 shows the execution time of the algorithms on the dataset *Wiki* and *English*. We have the following observations: (1) The execution time of GD-DCCS and GD-DCCS+ increases with k because the time cost for selecting d-CCs in GD-DCCS and

[2] http://string-db.org.
[3] http://cn.aminer.org.
[4] http://konect.uni-koblenz.de.
[5] http://snap.stanford.edu.

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. 18 Parameter configuration

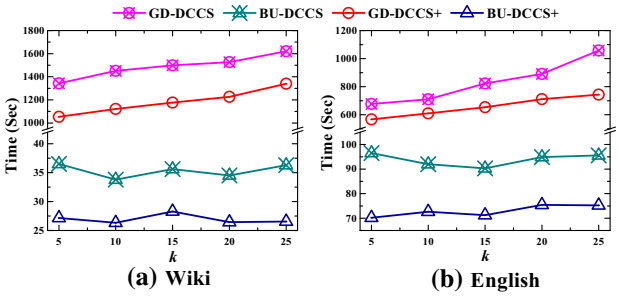


Fig. 19 Execution time versus parameter k (with small s)

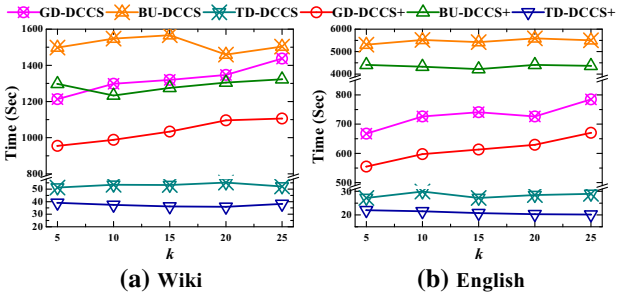


Fig. 20 Execution time versus parameter k (with large s)

GD-DCCS+ is proportional to k. (2) The execution time of BU-DCCS and BU-DCCS+ is insensitive to k. This is because the power of the pruning methods we adopted in BU-DCCS and BU-DCCS+ relies on the result cover size $|\mathsf{Cov}(\mathcal{R})|$ according to Eq. (1). However, as we will shown later, when k grows, $|\mathsf{Cov}(\mathcal{R})|$ increases insignificantly for small s, so k has little effects on the execution time of BU-DCCS and BU-DCCS+. (3) GD-DCCS+ and BU-DCCS+ run faster than GD-DCCS and BU-DCCS, respectively. This is because in these two algorithms, we adopt the optimization techniques to reduce the redundant examination of some vertices in d-CC computation, which in turn speeds up the algorithms. (4) BU-DCCS+ runs 1–2 orders of magnitude faster than GD-DCCS+. The main reason is that the pruning methods adopted by BU-DCCS+ reduce the search space of the DCCS problem by 80–90%. For the same reason, BU-DCCS also outperforms than GD-DCCS by 1–2 orders of magnitude.

We also examine the algorithms for large s and show results in Fig. 20. At this time, we also test the TD-DCCS and TD-DCCS+ algorithms. Except for the same observations of small s, we have the following new findings: 1) The execution time of TD-DCCS and TD-DCCS+ is also insensitive to k. This is also because the pruning methods adopted in

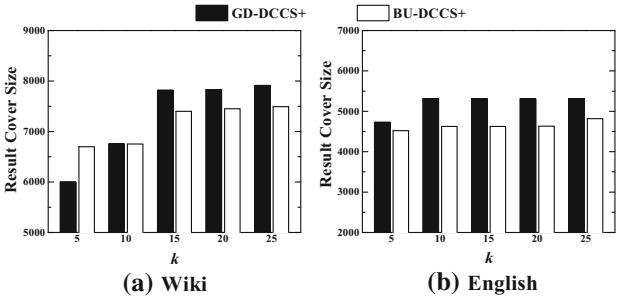


Fig. 21 Result cover size versus parameter k (with small s)

TD-DCCS and TD-DCCS+ rely on $|\mathsf{Cov}(\mathcal{R})|$. However, when k grows, $|\mathsf{Cov}(\mathcal{R})|$ increases insignificantly for large s, so k has little effects on the execution time of TD-DCCS and TD-DCCS+. 2) TD-DCCS+ runs faster than TD-DCCS. This is because we also apply the optimization techniques proposed in Sect. 7 in TD-DCCS+ to speed up the algorithm. 3) BU-DCCS and BU-DCCS+ are not efficient for large s. Sometimes, it is even worse than GD-DCCS and GD-DCCS+. This is because the size of the d-CCs significantly decreases for large s. BU-DCCS and BU-DCCS+ have to search down deep the search tree until the pruning techniques taking effects. Thus, BU-DCCS and BU-DCCS+ may search more d-CCs than GD-DCCS and GD-DCCS+. 4) TD-DCCS+ runs much faster than all the others. 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, lots of unpromising candidates d-CCs are pruned earlier in TD-DCCS+.

Result cover size w.r.t. parameter k We evaluate the cover size $|\mathsf{Cov}(\mathcal{R})|$ of the result $\mathcal{R}$ w.r.t. parameter k. Since the optimized GD-DCCS+, BU-DCCS+, and TD-DCCS+ algorithms always output the same results of the GD-DCCS, BU-DCCS, and TD-DCCS algorithms, respectively, we only present the results of the optimized algorithms. Figures 21 and 22 show the experimental results for small s and large s, respectively. We have two observations: (1) For all the algorithms, $|\mathsf{Cov}(\mathcal{R})|$ grows w.r.t. k, however insignificantly for $k \geq 15$. From another perspective, this implies that there exist substantial overlaps among d-CCs. To reduce redundancy, it is meaningful to find top-k diversified d-CCs on a multi-layer graph. (2) In most cases, the results of different 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+. Based on these observations, in the following experiments, we only examine GD-DCCS+ and BU-DCCS+ for small s and GD-DCCS+ and TD-DCCS+ for large s.

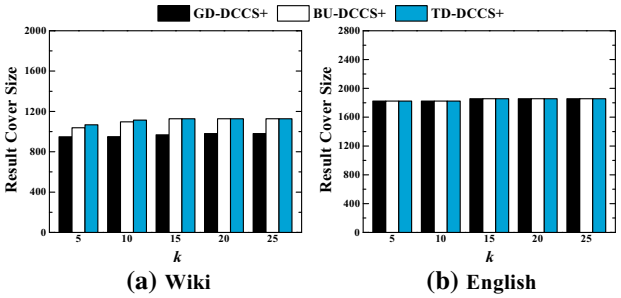


Fig. 22 Result cover size versus parameter k (with large s)

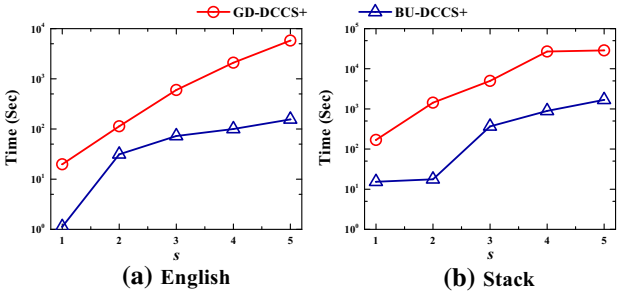


Fig. 23 Execution time versus parameter s (small)

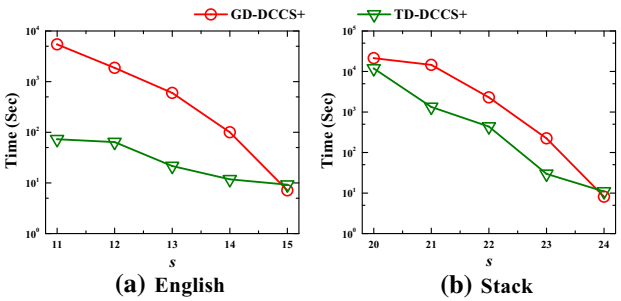


Fig. 24 Execution time versus parameter s (large)

Effects of parameter s We evaluate the effects of the parameter s on the performance of algorithms. By varying s, Figs. 23 and 24 show the execution time of the algorithms on the datasets *English* and *Stack* for small s and large s, respectively. We have the following observations: (1) For small s, the execution time of GD-DCCS+ and BU-DCCS+ 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) For large s, 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 $s \geq l(\mathcal{G})/2$. (3) For small s and large s, BU-DCCS+ and TD-DCCS+ runs much faster than GD-DCCS+.

Figures 25 and 26 show the effects of s on the results cover size for small s and large s, respectively. We find that: (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 decreases due to Property 3, so $\mathcal{R}$ cover less vertices. (2) The practical

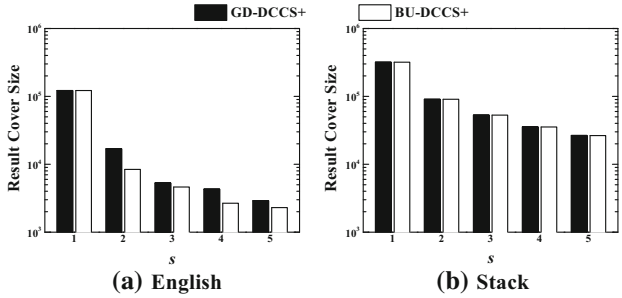


Fig. 25 Result cover size versus parameter s (small)

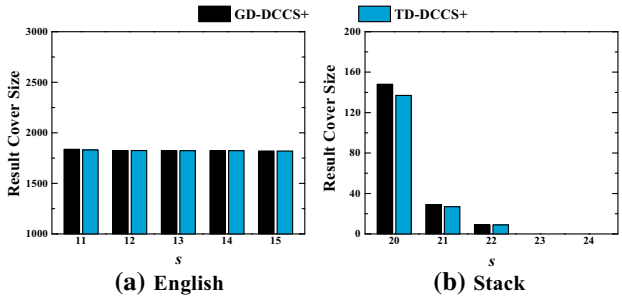


Fig. 26 Result cover size versus parameter s (large)

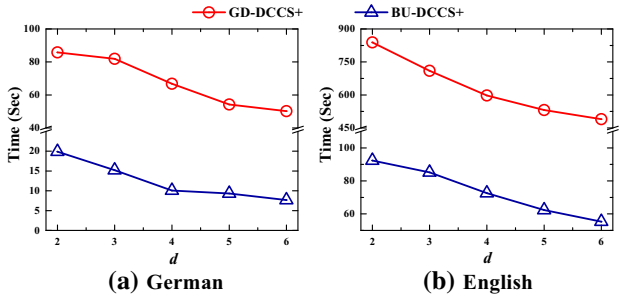


Fig. 27 Execution time versus parameter d (with small s)

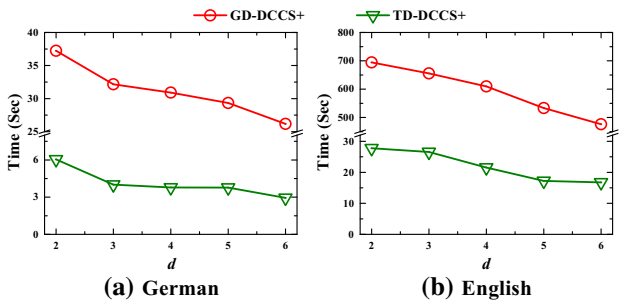


Fig. 28 Execution time versus parameter d (with large s)

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. 27 shows the execution time of BU-DCCS+ and GD-DCCS+ on datasets *German* and *English* for $s = 3$, and Fig. 28 shows the execution time of TD-DCCS+ and GD-DCCS+ on *German*

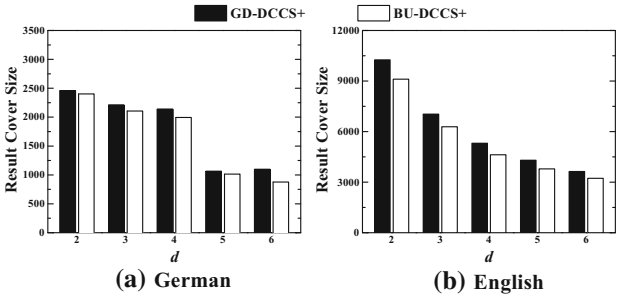


Fig. 29 Result cover size versus parameter d (with small s)

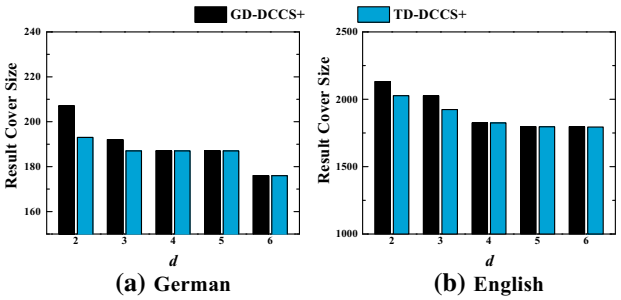


Fig. 30 Result cover size versus parameter d (with large s)

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) The size of the d-core on each layer decreases w.r.t d, so the algorithms spend less time on d-CC computation. Moreover, both BU-DCCS+ and TD-DCCS+ are much faster than GD-DCCS+.

Figures 29 and 30 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 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+.

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. Figure 31 shows the execution time of BU-DCCS+, TD-DCCS+, and GD-DCCS+ on the largest dataset *Stack* by varying p from 0.2 to 1.0 for both small and large s. All the algorithms scale linearly w.r.t. p because the time cost of computing d-CCs is linear to the vertex count.

Figure 32 shows the execution time of BU-DCCS+, TD-DCCS+, and GD-DCCS+ on *Stack* w.r.t. q for both small and

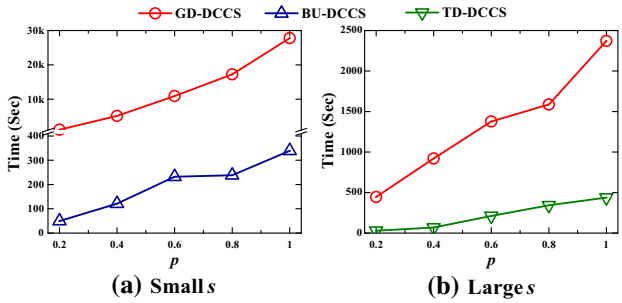


Fig. 31 Execution time versus parameter p

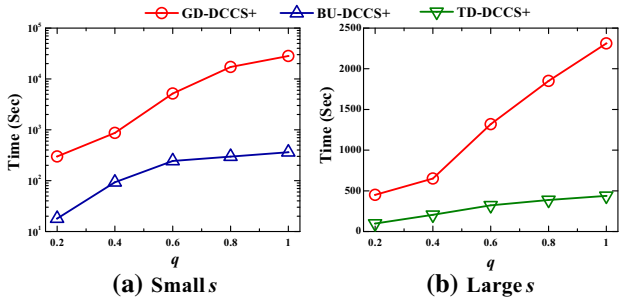


Fig. 32 Execution time versus parameter q

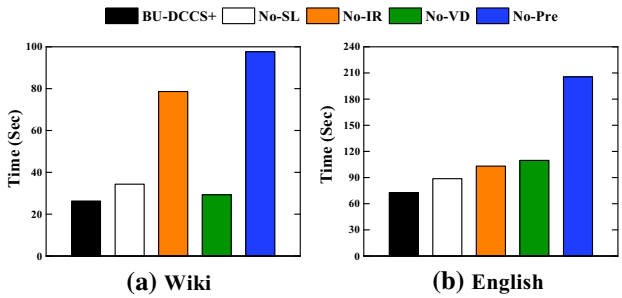


Fig. 33 Effects of preprocessing methods (with small s)

large s. We observed that: (1) The execution time of all algorithms grows with q. This is because the search space of the DCCS problem increases w.r.t. the layer number of the input multi-layer graph. (2) The execution time of GD-DCCS+ grows much faster than both 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 that of BU-DCCS+ and TD-DCCS+.

Effects of preprocessing methods We evaluate the effects of the preprocessing methods. Figures 33 and 34 show the comparison results for BU-DCCS+ and TD-DCCS+, respectively. Here, No-VD, No-SL, and No-IR mean the preprocessing method "vertex deletion," "sorting layers," and "result initialization" are disabled, respectively, and 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 ver-

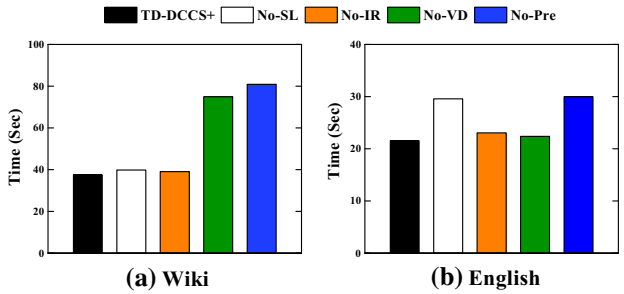


Fig. 34 Effects of preprocessing methods (with large s)

Graph $\mathcal{G}$	$d=2$	$d=3$	$d=4$	$d=5$	$d=6$
PPI	0.23	0.21	0.25	0.12	0.08
Author	0.57	0. 44	0.48	0.26	0.27
German	4.23	3.37	2.81	1.59	1.21
Wiki	5.09	3.68	3.54	2.73	1.95
English	20.89	17.61	14.97	12.23	10.07
Stack	89.23	80.21	73.79	59.44	56. 78

Fig. 35 Execution time of the BuildIndex procedure (in s)

ifies 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+.

Performance of index construction We also examine the performance of the BuildIndex procedure for constructing the index structure. Figure 35 shows the index construction time by varying d. We can see that: (1) The BuildIndex procedure runs very fast on all the datasets. In the DCCS algorithms, the fraction of index construction time is very small. According to Sect. 7.1, the index construction can be completed in linear time w.r.t. the graph size. (2) When d becomes larger, the execution time of BuildIndex decreases. This is because for larger d, the d-cores contain less vertices, so the index construction process runs much faster.

8.3 Comparison with quasi-clique-based algorithms

We compare our DCCS algorithms with three representative quasi-clique-based algorithms, namely Crochet [20], Cocain [32], and MiMAG [5], in terms of time efficiency and result similarity. 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 We set the parameters as follows: (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. Figure 36 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 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 similarity 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. 36, we have the following observations:

(1) *DCCS*($\mathcal{R}_D$) *versus Corchet* ($\mathcal{R}_C$) We have $|\mathsf{Cov}(\mathcal{R}_D)| > |\mathsf{Cov}(\mathcal{R}_C)|$, i.e., $\mathcal{R}_D$ 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 occurring on all layers, which are highly likely to be covered by the d-CCs found by DCCS.

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

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	1	1	1
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	1	1	1
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	**52.17**	451	1	1	1
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	**574.65**	3072	1	1	1

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

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	1	1	1
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	1	1	1
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	**52.17**	451	1	1	1
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	**574.65**	3072	1	1	1

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

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	1	1	1
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	1	1	1
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	**52.17**	451	1	1	1
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	**574.65**	3072	1	1	1

Fig. 36 Comparison between DCCS and quasi-clique-based algorithms Crochet, Cocain, and MiMAG

(2) *DCCS*($\mathcal{R}_D$) *versus Cocain*($\mathcal{R}_N$) The vertices covered by $\mathcal{R}_D$ significantly overlap 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*($\mathcal{R}_D$) *versus MiMAG*($\mathcal{R}_M$) The vertices covered by $\mathcal{R}_D$ largely overlap with the vertices covered by $\mathcal{R}_M$. In particular, 65%–85% vertices covered by $\mathcal{R}_D$ are also covered by $\mathcal{R}_M$; in reverse, 70%–88% vertices covered by $\mathcal{R}_M$ are also covered by $\mathcal{R}_D$.
(4) *MiMAG*($\mathcal{R}_M$) *versus Cocain*($\mathcal{R}_N$) *versus 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 close to $\mathcal{R}_D$ than $\mathcal{R}_N$.

In terms of both algorithm efficiency and result similarity, MiMAG outperforms Cocain and Corchet. For this reason, we present more details on evaluating the result quality between DCCS and MiMAG.

8.4 Evaluation of result quality

We introduced two applications in Sect. 1, namely biological module discovery and active co-author group extraction. Here, we evaluate the result quality of DCCS and MiMAG in these applications.

Biological module discovery We extract the protein–protein interaction networks of six organisms from the STRING DB. Each network contains eight layers representing the interactions detected by different methods. We use DCCS and MiMAG to find dense subgraphs on each network, which are closely related to *protein complexes*. We take the protein complexes recorded in the MIPS database[6] 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.

[6] http://mips.helmholtz-muenchen.de.

Organism	Metric	$min_s = 3$ MiMAG	$d = 2$ DCCS	$min_s = 4$ MiMAG	$d = 3$ DCCS	$min_s = 5$ MiMAG	$d = 4$ DCCS
E. Coli	CCR	63.2%	**69.1%**	55.6%	**58.2%**	50.9%	**51.3%**
	PCR	65.8%	**72.6%**	58.7%	**61.5%**	52.7%	**54.2%**
	PWCR	65.1%	**73.4%**	59.5%	**61.7%**	52.9%	**54.5%**
A. Lyrata	CCR	71.8%	**74.1%**	64.8%	**68.2%**	58.3%	**62.4%**
	PCR	77.6%	**83.5%**	71.7%	**75.3%**	62.5%	**67.8%**
	PWCR	73.9%	**75.2%**	68.5%	**70.8%**	61.7%	**63.1%**
D. Ananassae	CCR	65.4%	**67.1%**	63.3%	**63.7%**	59.0%	**59.1%**
	PCR	69.2%	**72.3%**	64.5%	**65.2%**	61.5%	**61.9%**
	PWCR	59.7%	**64.1%**	56.2%	**57.2%**	53.8%	**54.2%**
M. Bovis	CCR	75.3%	**86.0%**	70.5%	**78.4%**	62.1%	**70.7%**
	PCR	83.9%	**91.7%**	80.2%	**87.8%**	73.8%	**75.9%**
	PWCR	80.2%	**88.3%**	76.5%	**79.7%**	75.4%	**75.8%**
C. Remanei	CCR	59.3%	**66.8%**	55.6%	**62.4%**	52.0%	**58.5%**
	PCR	63.8%	**70.1%**	58.4%	**62.7%**	54.6%	**61.8%**
	PWCR	64.1%	**68.4%**	62.7%	**65.5%**	57.1%	**58.5%**
W. Glossinidia	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. 37 Comparison between DCCS and MiMAG in biological modules discovery

To evaluate the result quality under different parameter settings, we set $k = 10$, $s = 4$, $d = 2, 3, 4$, $\gamma = 0.8$ and the corresponding parameter min_s is set to 3, 4, 5, respectively. Figure 37 shows the experimental results for each setting. We have the following observations: (1) In terms of all the measures, DCCS outperforms MiMAG. As shown in Sect. 8.3, this is because the dense subgraphs found by DCCS cover more densely connected vertices than MiMAG. (2) As d (or min_s) increases, the values of all measures decrease. This is because as d (or min_s) increases, the results of DCCS and MiMAG both cover less vertices.

Active co-author group extraction We apply both DCCS and MiMAG on the dataset *Author* to extract active co-author groups. We set parameters $k = 10$, $s = 5$, $d = 3$, $\gamma = 0.8$, and $min_s = 4$. Figure 38 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 was not found by MiMAG.

To further show the difference of the dense subgraphs generated by DCCS and MiMAG, we present two explanatory examples in Fig. 39. The left one is a group of authors in data mining domain. MiMAG finds the famous scientists

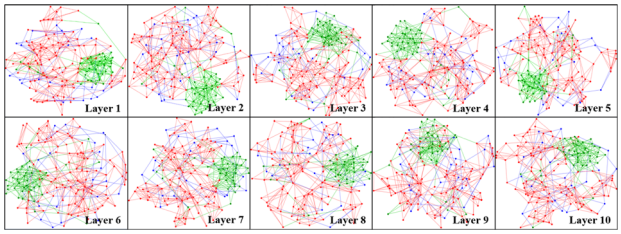


Fig. 38 Dense subgraphs found by DCCS and MiMAG on *Author*

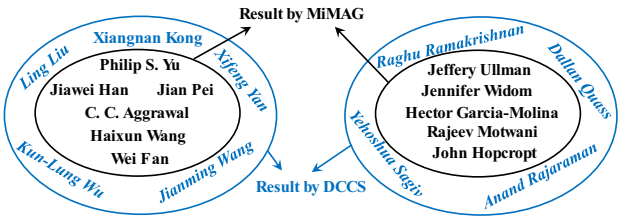


Fig. 39 Results found by DCCS and MiMAG

such as Philip S. Yu and Jiawei Han. However, DCCS also finds other scientists such as Xifeng Yan and Jianming Wang, who also have close collaborations with them. The right one is a group of authors in the database domain. We can also see that DCCS detects more researchers than MiMAG. The reason is that DCCS can find large-scale dense subgraphs covering more vertices than MiMAG.

From these two applications, we can see that our DCCS algorithm can find dense subgraphs missed by quasi-clique-based algorithms, so its result quality is much higher.

9 Related work

Dense subgraph mining is a fundamental graph mining task, which has been extensively studied on single-layer graphs. Recently, mining dense subgraphs on graphs with multiple types of edges has attracted much attention. A detailed survey can be found in [14]. Existing work can be categorized into two classes: dense subgraph mining on two-layer graphs and dense subgraph mining on general multi-layer graphs.

Dense subgraph mining on two-layer graphs Two-layer graph is a special multi-layer graph. In a two-layer graph, one layer represents physical link structures and the other represents conceptual connections between vertices derived from physical structures. The dense subgraph mining algorithms on two-layer graphs take both physical and conceptual connections into account. The algorithm in [17] finds dense subgraphs by expanding from initial seed vertices. The algorithm [21] adopts edge-induced matrix factorization. In [35], structural and attribute information is combined to form a unified distance measure, and a clustering algorithm is applied to detect dense subgraphs. In [29], structures and attributes are fused by a probabilistic model, and a model-based algorithm is proposed to find dense subgraphs. Other work on two-layer graphs includes the method based on correlation pattern mining [24] and graph merging [22]. All the algorithms are tailored to fit two-layer graphs. They only support the input where one layer represents physical connections and the other represents conceptual connections. Therefore, 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 many layers representing different types of edges between vertices. Dong [10] and Tang et al. [28] study dense subgraph mining using 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 are not scalable to large graphs. Alternatively, other work such as [5,20,32] focus on finding dense subgraph patterns by extending the quasi-clique notion defined on single-layer graphs. In [20,32], the algorithms find cross-graph quasi-cliques. In [5], the method is adapted to find diversified result to avoid redundancy. However, all these works have inherent limitations: (1) Quasi-clique-based methods are computationally costly; (2) the diameter of the discovered dense subgraphs is often very small. As verified by the experimental results in Sect. 8, the quasi-clique-based methods tend to miss large dense subgraphs.

d-CC versus cross-graph quasi-clique The d-CC and cross-graph quasi-clique are two different notions to characterize dense subgraphs on multi-layer graphs. As discussed in Sect. 1, each cross-graph quasi-clique is a motif containing a small number of cohesively connected vertices. Therefore, all dense subgraphs in a multi-layer graph are represented by a set of cross-graph quasi-cliques. However, the d-CC represents all dense subgraphs in a multi-layer graph by a macroscopic structure. The d-CC can be divided into multiple components. Each component is connected in some layers, which forms a dense subgraph in the multi-layer graph. The advantages of the d-CC notion are obvious. As shown in Sects. 8.3 and 8.4, our DCCS algorithms can fast detect larger dense subgraphs that cover most of the quasi-clique-based results.

We also discuss on some other related work.

Multi-layer graph analytics Multi-layer graphs are also called multi-view, multi-dimensional, or multi-attributed networks. Detailed surveys on analyzing multi-layer graphs can be found in [15,23]. In the literature, lots of algorithms have been proposed to address fundamental problems on multi-layer graphs, including the shortest path [4], the minimum spanning tree [8], graph clustering [19], and betweenness centrality [7,25]. Aside from analyzing algorithms, many work studies the applications of the multi-layer graphs in real-world scenarios, such as link prediction [33], topic detection [11], path planning [1], and congestion control [34].

Frequent subgraph pattern mining Given a set D of labeled graphs, frequent subgraph pattern mining discovers all subgraph patterns that are subgraph isomorphic to at least a fraction *minsup* of graphs in D (i.e., frequent) [30]. Our work is different from frequent subgraph pattern mining: (1) The graphs in D are labeled graphs. A vertex in a graph may not be identical to any vertex in other graphs. Hence, the graphs in D usually do not form a multi-layer graph. Inversely, a multi-layer graph is not necessary to be labeled. (2) 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 are not required to have the same link 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 [26]. 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 different types of vertices is 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 considers the cohesiveness of a vertex subset rather than its support.

d-cores on single-layer graphs The d-core notion is widely used to represent dense subgraphs on single-layer graphs. It has many useful properties and has been applied to community detection [18]. However, the d-core notion only considers density of but ignores support. In this paper, we propose the d-CC notion, which extends the d-core notion from two aspects: (1) considering both density and support of dense subgraphs; (2) inheriting the elegant properties of d-cores.

10 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 Nos. 61672189, 61532015, and 61732003.

A Missing proofs

Proof **(Property 1)** Suppose $C_L^d(\mathcal{G})$ is not unique. Let C_1 and C_2 be two distinct d-CCs of $\mathcal{G}$ w.r.t. L. Let $C = C_1 \cup C_2$. We have $C_1 \subset C$ and $C_2 \subset C$. On each layer $i \in L$, $G_i[C_1]$ is a subgraph of $G_i[C]$. Thus, for each vertex $v \in C$, we have $d_{G_i[C]}(v) \geq d_{G_i[C_1]}(v) \geq d$ for all $i \in L$. By the definition of d-CC, C is also a d-CC, so neither C_1 nor C_2 is maximum, which leads to a contradiction. Thus, $C_L^d(\mathcal{G})$ is unique. □

Proof **(Property 2)** Let $d_1, d_2 \in \mathbb{N}$ and $d_1 > d_2$. For each vertex $v \in C_L^{d_1}(\mathcal{G})$, we have $d_{G_l[C_L^{d_1}(\mathcal{G})]}(v) \geq d_1 > d_2$ for every layer number $l \in L$. By the definition of d-CC, $C_L^{d_1}(\mathcal{G}) \subseteq C_L^{d_2}(\mathcal{G})$. Thus, the property holds. □

Proof **(Property 3)** For each vertex $v \in C_{L'}^d(\mathcal{G})$, we have $d_{G_l[C_{L'}^d(\mathcal{G})]}(v) \geq d$ for each layer number $l \in L$. Based on the definition of d-CC, we have $C_{L'}^d(\mathcal{G}) \subseteq C_L^d(\mathcal{G})$. Hence, the property holds. □

Proof **(Lemma 1)** It is clear that $L_1 \subseteq L_1 \cup L_2$ and $L_2 \subseteq L_1 \cup L_2$. By Property 3, we have $C_{L_1 \cup L_2}^d(\mathcal{G}) \subseteq C_{L_1}^d(\mathcal{G})$ and $C_{L_1 \cup L_2}^d(\mathcal{G}) \subseteq C_{L_2}^d(\mathcal{G})$. Thus, $C_{L_1 \cup L_2}^d(\mathcal{G}) \subseteq C_{L_1}^d(\mathcal{G}) \cap C_{L_2}^d(\mathcal{G})$. □

Proof **(Lemma 2)** Let $C_{L'}^d(\mathcal{G})$ be a descendant of $C_L^d(\mathcal{G})$. We have $L \subseteq L'$. By Property 3, we have $C_{L'}^d(\mathcal{G}) \subseteq C_L^d(\mathcal{G})$. Thus, $\mathsf{Cov}((\mathcal{R} - \{C^*(\mathcal{R})\}) \cup \{C_{L'}^d(\mathcal{G})\}) \subseteq \mathsf{Cov}((\mathcal{R} - \{C^*(\mathcal{R})\}) \cup \{C_L^d(\mathcal{G})\})$. Obviously, if we have $|\mathsf{Cov}((\mathcal{R} - \{C^*(\mathcal{R})\}) \cup \{C_L^d(\mathcal{G})\})| < \left(1 + \frac{1}{k}\right)|\mathsf{Cov}(\mathcal{R})|$, we must have $|\mathsf{Cov}((\mathcal{R} - \{C^*(\mathcal{R})\}) \cup \{C_{L'}^d(\mathcal{G})\})| < \left(1 + \frac{1}{k}\right)|\mathsf{Cov}(\mathcal{R})|$. Thus, $C_{L'}^d(\mathcal{G})$ cannot satisfy Eq. (1), which means none of the descendants of $C_L^d(\mathcal{G})$ satisfies Eq. (1). □

Proof **(Lemma 3)** For any subset $D \subseteq L_P$ such that $|D| = s - |L|$, since $I_D = \cap_{i \in D} C^d(G_i)$, we have $I_D \subseteq C^d(G_i)$ for each $i \in D$. Consequently, for each vertex $v \in I_D$, we have $v \in C^d(G_i)$ for each $i \in D$. That is, v must be contained in at least $s - |L|$ d-cores on the layers in L_P, so $v \in I$. Therefore, we have $I_D \subseteq I$. □

Proof **(Lemma 4)** Let $C_S^d(\mathcal{G})$ be a descendant of $C_L^d(\mathcal{G})$ such that $|S| = s$. Let $D = S - L$ and $I_D = \cap_{i \in D} C^d(G_i)$. By Lemma 1, we have $C_S^d(\mathcal{G}) \subseteq C_L^d(\mathcal{G}) \cap I_D$. Since $I_D \subseteq I$ according to Lemma 3, we have $C_S^d(\mathcal{G}) \subseteq C_L^d(\mathcal{G}) \cap I$. For ease of presentation, let $C = C_L^d(\mathcal{G}) \cap I$. We illustrate the relationships between $\mathsf{Cov}(\mathcal{R})$, $C^*(\mathcal{R})$ and C in Fig. 40 with seven disjoint subsets A, B, D, E, F, G, and H. We have $|\mathsf{Cov}(\mathcal{R})| = |A| + |B| + |D| + |F| + |G| + |H|$, $|C^*(\mathcal{R})| = |B| + |D| + |G| + |H|$, $|C| = |D| + |E| + |F| + |G|$, $|\Delta(\mathcal{R}, C^*(\mathcal{R}))| = |D| + |H|$.

Since $|C| < \frac{1}{k}|\mathsf{Cov}(\mathcal{R})| + |\Delta(\mathcal{R}, C^*(\mathcal{R}))|$, we have $|D| + |E| + |F| + |G| < \frac{1}{k}(|A| + |B| + |D| + |F| + |G| + |H|) + |D| + |H|$. Thus,

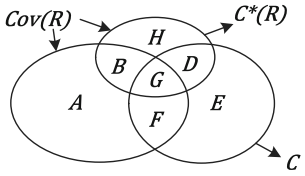


Fig. 40 Relationships between $\mathsf{Cov}(\mathcal{R})$, $C^*(\mathcal{R})$, and C

$$
\begin{aligned}
&|\mathsf{Cov}((\mathcal{R} - \{C^*(\mathcal{R})\}) \cup \{C\})| \\
&= |A| + |B| + |D| + |E| + |F| + |G| \\
&< \frac{1}{k}(|A| + |B| + |D| + |G| + |F| + |H|) + |A| \\
&+ |B| + |D| + |H| \\
&\leq \left(1 + \frac{1}{k}\right)(|A| + |B| + |D| + |G| + |F| + |H|) \\
&= \left(1 + \frac{1}{k}\right)|\mathsf{Cov}(\mathcal{R})|.
\end{aligned}
$$

Since $C^d_S(\mathcal{G}) \subseteq C$, we have $\mathsf{Cov}((\mathcal{R} - \{C^*(\mathcal{R})\}) \cup \{C^d_S(\mathcal{G})\}) \subseteq \mathsf{Cov}((\mathcal{R} - \{C^*(\mathcal{R})\}) \cup \{C\})$. Then, we have $|\mathsf{Cov}((\mathcal{R} - \{C^*(\mathcal{R})\}) \cup \{C^d_S(\mathcal{G})\})| \leq |\mathsf{Cov}((\mathcal{R} - \{C^*(\mathcal{R})\}) \cup \{C\})| < \left(1 + \frac{1}{k}\right)|\mathsf{Cov}(\mathcal{R})|$. Thus, the lemma thus holds. □

Proof **(Lemma 5)** By the definitions of d-CC and d-core, we have $C^d(G_j) = C^d_{\{j\}}(\mathcal{G})$. By Lemma 1, we have $C^d_{L\cup\{j\}}(\mathcal{G}) \subseteq C^d_L(\mathcal{G}) \cap C^d(G_j)$. Let $C = C^d_L(\mathcal{G}) \cap C^d(G_j)$, in similar to the proof of Lemma 4, if $|C| < \frac{1}{k}|\mathsf{Cov}(\mathcal{R})| + |\Delta(\mathcal{R}, C^*(\mathcal{R}))|$, we have $\mathsf{Cov}((\mathcal{R} - \{C^*(\mathcal{R})\}) \cup \{C^d_{L\cup\{j\}}(\mathcal{G})\}) \subseteq \mathsf{Cov}((\mathcal{R} - \{C^*(\mathcal{R})\}) \cup \{C\})$. Then, we must have $|\mathsf{Cov}((\mathcal{R} - \{C^*(\mathcal{R})\}) \cup \{C^d_{L\cup\{j\}}(\mathcal{G})\})| \leq |\mathsf{Cov}((\mathcal{R} - \{C^*(\mathcal{R})\}) \cup \{C\})| < \left(1 + \frac{1}{k}\right)|\mathsf{Cov}(\mathcal{R})|$. The lemma thus holds. □

Proof **(Lemma 6)** Since $L \subseteq S$, we have $L \cup \{j\} \subseteq S \cup \{j\}$. According to Property 3, we have $C^d_{S\cup\{j\}}(\mathcal{G}) \subseteq C^d_{L\cup\{j\}}(\mathcal{G})$. Therefore, $\mathsf{Cov}((\mathcal{R} - \{C^*(\mathcal{R})\}) \cup \{C^d_{S\cup\{j\}}(\mathcal{G})\}) \subseteq \mathsf{Cov}((\mathcal{R} - \{C^*(\mathcal{R})\}) \cup \{C^d_{L\cup\{j\}}(\mathcal{G})\})$. Since $C^d_{L\cup\{j\}}(\mathcal{G})$ does not satisfy Eq. (1), we must have $|\mathsf{Cov}((\mathcal{R} - \{C^*(\mathcal{R})\}) \cup \{C^d_{S\cup\{j\}}(\mathcal{G})\})| < \left(1 + \frac{1}{k}\right)|\mathsf{Cov}(\mathcal{R})|$. Thus, the lemma holds. □

Proof **(Lemma 7)** According to the usage of potential vertex sets, for any descendant $C^d_{L'}(\mathcal{G})$ of $C^d_L(\mathcal{G})$ with $|L'| = s$, we have $C^d_{L'}(\mathcal{G}) \subseteq U^d_L(\mathcal{G})$. In similar to the proof of Lemma 2, the lemma holds. □

Proof **(Lemma 8)** Similar to the proof of Lemma 5, we have $|\mathsf{Cov}((\mathcal{R} - \{C^*(\mathcal{R})\}) \cup \{U^d_{L-\{j\}}(\mathcal{G})\})| < \left(1 + \frac{1}{k}\right)|\mathsf{Cov}(\mathcal{R})|$. According to the usage of potential sets, for any descendant $C^d_{L'}(\mathcal{G})$ of $C^d_L(\mathcal{G})$ with $|L'| = s$, we have $C^d_{L'}(\mathcal{G}) \subseteq U^d_L(\mathcal{G})$. Thus, we must have $|\mathsf{Cov}((\mathcal{R} - \{C^*(\mathcal{R})\}) \cup \{C^d_{L'}(\mathcal{G})\})| \leq |\mathsf{Cov}((\mathcal{R} - \{C^*(\mathcal{R})\}) \cup \{U^d_{L-\{j\}}(\mathcal{G})\})| < \left(1 + \frac{1}{k}\right)|\mathsf{Cov}(\mathcal{R})|$. Thus, the lemma holds. □

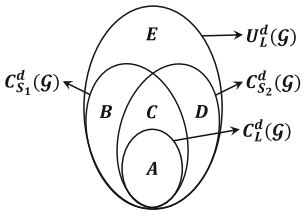


Fig. 41 Relationships between $C^d_L(\mathcal{G})$, $C^d_{S_1}(\mathcal{G})$, $C^d_{S_2}(\mathcal{G})$, and $U^d_L(\mathcal{G})$

Proof **(Lemma 9)** We illustrate the relationships between $C^d_L(\mathcal{G})$, $C^d_{S_1}(\mathcal{G})$, $C^d_{S_2}(\mathcal{G})$, and $U^d_L(\mathcal{G})$ in Fig. 41 with five disjoint subsets A, B, C, D, and E. We have $|C^d_{S_1}(\mathcal{G})| = |A| + |B| + |C|$, $|C^d_{S_2}(\mathcal{G})| = |A| + |C| + |D|$, $|U^d_L(\mathcal{G})| = |A| + |B| + |C| + |D| + |E|$, $|C^d_{S_1}(\mathcal{G}) \cap C^d_{S_2}(\mathcal{G})| = |A|$.

Since $C^d_{S_1}(\mathcal{G})$ can update $\mathcal{R}$, Lemma 5 implies that $|C^d_{S_1}(\mathcal{G})| \geq \frac{1}{k}|\mathsf{Cov}(\mathcal{R})| + |\Delta(\mathcal{R}, C^*(\mathcal{R}))|$. Let $\mathcal{R}'$ be the resulting $\mathcal{R}$ after updating $\mathcal{R}$ with $C^d_{S_1}(\mathcal{G})$. We have $|\mathsf{Cov}(\mathcal{R}')| \geq \left(1 + \frac{1}{k}\right)|\mathsf{Cov}(\mathcal{R})|$.

Suppose $C^d_{S_2}(\mathcal{G})$ can update $\mathcal{R}'$ again, then we have $|\mathsf{Cov}((\mathcal{R}' - \{C^*(\mathcal{R}')\}) \cup \{C^d_{S_2}(\mathcal{G})\})| \geq \left(1 + \frac{1}{k}\right)|\mathsf{Cov}(\mathcal{R}')| \geq \left(\frac{1}{k} + \frac{1}{k^2}\right)|\mathsf{Cov}(\mathcal{R})|$. Since $A \cup C \subset C^d_{S_2}(\mathcal{G})$, $\mathsf{Cov}((\mathcal{R}' - \{C^*(\mathcal{R}')\}) \cup \{C^d_{S_2}(\mathcal{G})\}) = \mathsf{Cov}(\mathcal{R}') - \Delta(\mathcal{R}', C^*(\mathcal{R}')) + D \subseteq \mathsf{Cov}(\mathcal{R}') + D$.

Putting them together, we have $|\mathsf{Cov}(\mathcal{R}')| + |D| \geq |\mathsf{Cov}((\mathcal{R}' - \{C^*(\mathcal{R}')\}) \cup \{C^d_{S_2}(\mathcal{G})\})| \geq \left(1 + \frac{1}{k}\right)|\mathsf{Cov}(\mathcal{R}')|$. That means $|D| \geq \frac{1}{k}|\mathsf{Cov}(\mathcal{R}')|$. Thus, for $U^d_L(\mathcal{G})$, we have

$$
\begin{aligned}
|U^d_L(\mathcal{G})| &= |A| + |B| + |C| + |D| + |E| \geq |C^d_{S_1}(\mathcal{G})| + |D| \\
&\geq \frac{1}{k}|\mathsf{Cov}(\mathcal{R})| + |\Delta(\mathcal{R}, C^*(\mathcal{R}))| + \frac{1}{k}|\mathsf{Cov}(\mathcal{R}')| \\
&= \left(\frac{1}{k} + \frac{1}{k^2}\right)|\mathsf{Cov}(\mathcal{R})| + |\Delta(\mathcal{R}, C^*(\mathcal{R}))| \\
&\quad + \frac{1}{k}|\mathsf{Cov}(\mathcal{R})| \\
&\geq \left(\frac{1}{k} + \frac{1}{k^2}\right)|\mathsf{Cov}(\mathcal{R})| \\
&\quad + \left(1 + \frac{1}{k}\right)|\Delta(\mathcal{R}, C^*(\mathcal{R}))|.
\end{aligned}
$$

The last equation holds due to the pigeonhole principle. For each $C' \in \mathcal{R}$, we must have $|\Delta(\mathcal{R}, C')| \leq \frac{1}{k}|\mathsf{Cov}(\mathcal{R})|$. Now, $|U^d_L(\mathcal{G})|$ contradicts with Eq. (2). Thus, if $U^d_L(\mathcal{G})$ satisfies Eq. (2), $C^d_{S_2}(\mathcal{G})$ cannot update $\mathcal{R}$ any more. □

Proof **(Lemma 10)** We prove that if there not exists a candidate path in the index to w, w certainly does not exist in $C^d_L(\mathcal{G})$.

First, for each vertex v in the lowest level, if $L \not\subseteq L(v)$, there must exist a layer number $j \in L$ such that $v \notin C^d(G_j)$. By Lemma 1, we must have $v \notin C^d_L(\mathcal{G})$. Thus, we can remove

all such vertices from the graph and the index. After that, we consider each vertex w in the next level of the lowest level. At this time, all of w's neighbors u in the lowest level such that $L \nsubseteq L(u)$ have already been removed from the graph. Thus, vertex w has the same neighbors as we build the index. If $L \nsubseteq L(w)$, there must exist a layer number $j' \in L$ such that $w \notin C^d(G_{j'})$. By Lemma 1, w cannot be contained in $C^d_L(\mathcal{G})$. We can continue this process level by level. This implies that all the vertices that do not satisfy this condition cannot exist in $C^d_L(\mathcal{G})$. □

Proof **(Lemma 11)** We have $C^d_L(\mathcal{G}) \subseteq Y$ by the definition of Y. For each vertex v, if $v \in \bigcup_{h=0}^{|L|-1} I_h$, the support of v is less than $|L|$. Thus, v is unlikely to exist in a d-CC on at least $|L|$ layers. Therefore, we must have $v \in \bigcup_{h=|L|}^{l(\mathcal{G})} I_h$. Thus, the lemma holds. □

Proof **(Lemma 12)** We prove this lemma by simply contradiction. Suppose there exists a vertex $v \notin C^d_L(\mathcal{G})$ and v is not set to be discarded after the FastdCC procedure. We must have $d_i^+(v) \geq d$ for each $i \in L$. Otherwise, v must set to be discarded at line 2 of the ProcessUnd procedure or line 5 of the ProcessDis procedure. At this time, since v only connects to undetermined or existing vertices, we have $d_{G_i}(v) = d_i^+(v) \geq d$ for each $i \in L$. Therefore, we have $v \in C^d_L(\mathcal{G})$, which leads to a contradiction. Thus, the lemma holds. □

Proof **(Lemma 13)** To analyze the time complexity of FastdCC procedure, we first analyze the cases when an edge can be accessed as follows:

(1) At line 4 of the FastdCC procedure, when computing $d_i^+(v)$ and $d_i^*(v)$ of all $i \in L$ for each vertex v, each edge (u, v) on a layer $i \in L$ will be accessed exactly once.
(2) At line 8 of the ProcessUnd procedure, when vertex u accesses a vertex v on a higher level, each edge (u, v) on a layer $i \in L'$ will be accessed exactly once.
(3) At line 3 of the ProcessDis procedure, when updating $d_i^+(u)$, the edge (u, v) on a layer $i \in L$ will be accessed. Note that the edge (u, v) on a layer $i \in L$ will be accessed only once. This is because, when updating $d_i^+(u)$, v has already been set to be discarded. The state of a discarded vertex will never change. Thus, v will never have chance to visit u any more. Meanwhile, since v is discarded, u also will not visit vertex v afterward.
(4) At line 3 of the ProcessEst procedure, when updating $d_i^*(u)$, the edge (u, v) on a layer $i \in L$ will be accessed. In similar, at this time, v has already been set to be existing. The state of an existing vertex will also never change. Thus, the edge (u, v) on a layer $i \in L$ will also be accessed only once.

Thus, the total edge access time is $O(4\sum_{i \in L'} |E_i(\mathcal{G})|) = O(4m')$.

Next, we analyze the time cost on comparing $d_i^+(v)$ and $d_i^*(v)$ on each vertex v w.r.t. d as follows.

(1) At line 1 and line 3 of the ProcessUnd procedure, we compare $d_i^+(v)$ and $d_i^*(v)$ w.r.t. d for each vertex v which is set to undetermined at the first time. The time cost for vertex v is $O(2l)$. Thus, the total time cost for all vertices is $O(2nl)$.
(2) At line 4 of the ProcessDis procedure which compares $d_i^+(v)$ w.r.t. d, since the comparison can be involved in the updating of $d_i^+(v)$ at line 3, the comparison times equals to the number of the edge access times. As we analyzed earlier, each edge on each level is accessed at most once, so the total time cost is $O(m')$.
(3) At line 4 of the ProcessEst procedure which compares $d_i^*(v)$ w.r.t. d, the comparison can also be involved in the updating of $d_i^*(v)$ at line 3. In similar, the comparison times equals to the number of the edge access times. The total time cost is $O(m')$.

Meanwhile, the time cost to set the states of each vertex is at most $O(4n)$ since there are only four states in the procedure. Putting them together, the time complexity of the FastdCC procedure is $O(2nl + 6m' + 4n) = O(nl + m')$. □

Proof **(Theorem 1)** Given a collection of sets $\mathcal{F} = \{C_1, C_2, \ldots, C_n\}$ and $k \in \mathbb{N}$, the max-k-cover problem is to find a subset $\mathcal{R} \subseteq \mathcal{F}$ such that $|\mathcal{R}| = k$ and that $|\mathsf{Cov}(\mathcal{R})|$ is maximized. The max-k-cover problem has been proved to be NP-complete unless P = NP [2].

It is easy to show that the DCCS problem is in NP. We prove the theorem by reduction from the max-k-cover problem in polynomial time. Given an instance $(\mathcal{F}, k)$ of the max-k-cover problem, we first construct a multi-layer graph $\mathcal{G}$. The vertex set of $\mathcal{G}$ is $\bigcup_{i=1}^{n} C_i$. There are n layers in $\mathcal{G}$. An edge (u, v) exists on layer i if and only if $u, v \in C_i$ and $u \neq v$. Then, we construct an instance of the DCCS problem $(\mathcal{G}, d, s, k)$, where $d = 1$ and $s = 1$. The result of the DCCS problem instance $(\mathcal{G}, d, s, k)$ is exactly the result of the max-k-cover problem instance $(\mathcal{F}, k)$. The reduction can be done in polynomial time. Thus, the DCCS problem is NP-complete. □

To prove Theorems 2 and 3, we first state the following claim. The correctness of the claim has been proved in [2].

Claim Let $\mathcal{F} = \{C_1, C_2, \ldots, C_n\}$ and $k \in \mathbb{N}$. Let $\mathcal{R}^*$ the subset of $\mathcal{F}$ such that $|\mathcal{R}^*| = k$ and $|\mathsf{Cov}(\mathcal{R}^*)|$ is maximized. Let $\mathcal{R} \subseteq \mathcal{F}$ be a set obtained in the following way. Initially, $\mathcal{R} = \emptyset$. We repeat taking an element C out of $\mathcal{F}$ randomly and updating $\mathcal{R}$ with C according to the two rules specified in Sect. 5.1 until $\mathcal{F} = \emptyset$. Finally, we have $|\mathsf{Cov}(\mathcal{R})| \geq \frac{1}{4}|\mathsf{Cov}(\mathcal{R}^*)|$.

Proof **(Theorem 3)** Note that the BU-DCCS algorithm uses the same procedure described in Claim A to update $\mathcal{R}$ except that some pruning techniques are applied as well. Therefore, we only need to show that the pruning techniques will not affect the approximation ratio stated in Claim A. Let C be a d-CC pruned by a pruning method and D_C be the set of descendant candidate d-CCs of C in the search tree. For all $C' \in D_C$, according to Lemmas 2, 4, or 5, C' must not update $\mathcal{R}$. By Claim A, candidate d-CCs can be taken in an arbitrary order without affecting the approximation ratio. Therefore, we can safely ignore all the d-CCs in D_C without affecting the quality of $\mathcal{R}$. Finally, we have $|\mathsf{Cov}(\mathcal{R})| \geq \frac{1}{4}|\mathsf{Cov}(\mathcal{R}^*)|$. Thus, the theorem holds. □

Proof **(Theorem 4)** The TD-DCCS algorithm uses the same procedure described in Claim A to update $\mathcal{R}$ and applies some pruning techniques in addition. By the same arguments in the proof of Theorem 2, this theorem holds. □

B The Update procedure

We present the Update procedure in Fig. 42. The input includes the set $\mathcal{R}$ of temporary top-k diversified d-CCs, a newly generated d-CC C and $k \in \mathbb{N}$. For each d-CC $C' \in \mathcal{R}$, we store both C' and the size $|\Delta(\mathcal{R}, C')|$. To facilitate fast updating of $\mathcal{R}$, we build some auxiliary data structures. Specifically, we store $\mathcal{R}$ in two hash tables M and H. For each entry in M, the key of the entry is a vertex v, and the value of the entry is $M[v] = \{C' | C' \in \mathcal{R}, v \in C'\}$, that is, the set of d-CCs $C' \in \mathcal{R}$ containing vertex v. For each entry in H, the key of the entry is an integer i, and the value of the entry $H[i]$ is the set of d-CCs $C' \in R$ such that $|\Delta(\mathcal{R}, C')| = i$. Obviously, $C^*(\mathcal{R})$ can be easily obtained from H by retrieving the entry of H indexed by the smallest key.

Given the temporary result set $\mathcal{R}$ and a new d-CC C, the procedure relies on three key operations to update $\mathcal{R}$, namely $\mathsf{Size}(\mathcal{R}, C)$ that returns the size $|\mathsf{Cov}((\mathcal{R} - \{C^*(\mathcal{R})\}) \cup \{C\})|$, $\mathsf{Delete}(\mathcal{R})$ that removes $C^*(\mathcal{R})$ from $\mathcal{R}$, and $\mathsf{Insert}(\mathcal{R}, C)$ that inserts C to $\mathcal{R}$. We describe these procedures as follows.

Operation $\mathsf{Size}(\mathcal{R}, C)$ Note that $\mathsf{Cov}((\mathcal{R} - \{C^*(\mathcal{R})\}) \cup \{C\}$ can be decomposed into three disjoint subsets $\mathsf{Cov}(\mathcal{R} - \{C^*(\mathcal{R})\})$, $C - \mathsf{Cov}(\mathcal{R})$ and $C \cap \Delta(\mathcal{R}, C^*(\mathcal{R}))$. In the beginning, we can obtain $C^*(\mathcal{R})$ and $|\Delta(\mathcal{R}, C^*(\mathcal{R}))|$ from H (line 1) and initialize the counter c to 0 (line 1). For each vertex $v \in C$, if v is not a key in M, we have $v \in C - \mathsf{Cov}(\mathcal{R})$, so we increase c by 1 (line 5). Otherwise, if $v \in C^*(\mathcal{R})$ and $M[v]$ only contains $C^*(\mathcal{R})$, c is also increased by 1 (line 7) since $v \in C \cap \Delta(\mathcal{R}, C^*(\mathcal{R}))$. Since $|\mathsf{Cov}(\mathcal{R} - \{C^*(\mathcal{R})\})|$ is equal to $\mathsf{size}(M) - |\Delta(\mathcal{R}, C^*(\mathcal{R}))|$, we accumulate $\mathsf{size}(M) - |\Delta(\mathcal{R}, C^*(\mathcal{R}))|$ to c (line 8) and return c as the result (line 9).

```
Procedure Update(R, C, k)
1: if |R| < k then
2:     Insert(R, C)
3: else
4:     |Cov(R)| = size(M)
5:     if Size(R, C) ≥ (1 + 1/k)|Cov(R)| then
6:         Delete(R)
7:         Insert(R, C)
Procedure Size(R, C)
1: obtain C*(R) and |Δ(R, {C*(R)}| from H
2: c ← 0
3: for each vertex v ∈ C do
4:     if v is not a key in M then
5:         c ← c + 1
6:     else if v ∈ C*(R) and size(M[v]) = 1 then
7:         c ← c + 1
8: c ← c + size(M) − |Δ(R, C*(R))|
9: return c
Procedure Delete(R)
1: remove C*(R) from H
2: for each vertex v ∈ C*(R) do
3:     remove C*(R) from M[v]
4:     if size(M[v]) then
5:         let C' be the element in M[v]
6:         move C' in H from H[|Δ(R, C')|] to H[|Δ(R, C')| + 1]
7:         increase |Δ(R, C')| by 1
8:     else if size(M[v]) = 0 then
9:         remove v from M
Procedure Insert(R, C)
1: add C into R
2: set |Δ(R, C)| to 0
3: for each vertex v ∈ C do
4:     if v is not a key in M then
5:         add v into M
6:         insert C into M[v]
7:         increase |Δ(R, C)| by 1
8:     else
9:         if size(M[v]) = 1 then
10:            let C' be the element in M[v]
11:            move C' in H from H[|Δ(R, C')|] to H[|Δ(R, C')| − 1]
12:            decrease |Δ(R, C)| by 1
13:        insert C into M[v]
14: insert C into H based on |Δ(R, C)|
```

Fig. 42 The Update, Size, Delete, and Insert procedure

Operation $\mathsf{Delete}(\mathcal{R})$ First, we retrieve $C^*(\mathcal{R})$ from H (line 1). For each vertex $v \in C^*(\mathcal{R})$, $C^*(\mathcal{R})$ is removed from $M[v]$ (line 3). Note that, if $M[v]$ contains a single element C' after removing $C^*(\mathcal{R})$, v is a vertex only covered by C'. Therefore, we move C' from $H[|\Delta(\mathcal{R}, C')|]$ to $H[|\Delta(\mathcal{R}, C')| + 1]$ (line 6) and increase $|\Delta(\mathcal{R}, C')|$ by 1 (line 7). If $M[v]$ is empty, v is not covered by $\mathcal{R}$, so v is removed from M (line 9).

Operation $\mathsf{Insert}(\mathcal{R}, C)$ First, we insert C to $\mathcal{R}$ (line 1) and then set $|\Delta(\mathcal{R}, C)|$ to 0 (line 2). For each vertex $v \in C$, if v is not a key in M, we insert an entry with key v and value C to hash table M (lines 5–6). At this moment, v is only covered by C, so $|\Delta(\mathcal{R}, C)|$ is increased by 1 (line 7). If v is a key in M, C can be directly inserted to $M[v]$ (line 12). Note that, if $M[v]$ contains a single element C' before insertion, v will not be covered only by C' after inserting C, so C' is moved in H from $H[|\Delta(\mathcal{R}, C')|]$ to $H[|\Delta(\mathcal{R}, C')| - 1]$ (line 11), and $|\Delta(\mathcal{R}, C')|$ is decreased by 1 (line 12). After updating M, we obtain $|\Delta(\mathcal{R}, C)|$ and insert C to H accordingly (line 14).

Putting them altogether, we have the Update procedure. If $|\mathcal{R}| < k$, we directly insert C to $\mathcal{R}$ (line 2). If $|\mathcal{R}| \geq k$, the $\mathsf{Size}(\mathcal{R}, C)$ procedure is invoked to check if C satisfies Rule 2

(line 5). If so, $\mathcal{R}$ is updated with C by invoking Delete($\mathcal{R}$) and Insert($\mathcal{R}$, C) (lines 6–7).

Complexity analysis First we analyze the time complexity of the Update procedure. Assume that an entry can be inserted to or deleted from a hash table in constant time. Thus, the time complexity of Size($\mathcal{R}$, C), Delete($\mathcal{R}$), and Insert($\mathcal{R}$, C) is $O(|C|)$, $O(|C^*(\mathcal{R})|)$ and $O(|C|)$, respectively. Consequently, the time complexity of Update is obviously $O(\max\{|C|, |C^*(\mathcal{R})|\})$.

The space cost for storing the result set $\mathcal{R}$ and maintaining the hash table M is $O(\sum_{C' \in \mathcal{R}} |C'|)$, and the space cost for storing $|\Delta(\mathcal{R}, C')|$ and maintaining the hash table H is $O(k)$. Thus, the space complexity of Update is $O(2 \sum_{C' \in \mathcal{R}} |C_j| + 2k) = O(\sum_{C' \in \mathcal{R}} |C'|)$.

References

1. Abdel-Rahim, A., Oman, P., Johnson, B.K., Sadiq R.A.: Assessing surface transportation network component criticality: a multi-layer graph-based approach. In: IEEE Intelligent Transportation Systems Conference, pp. 1000–1003 (2007)
2. Ausiello, G., Boria, N., Giannakos, A., Lucarelli, G., Paschos, V.T.: Online maximum k-coverage. In: International Conference on Fundamentals of Computation Theory, pp. 181–192 (2011)
3. Batagelj, V., Zaversnik, M.: An O(m) algorithm for cores decomposition of networks. Comput. Sci. **1**(6), 34–37 (2003)
4. Bilbro, G.L.: Solution of the recirculant multilayer graph problem using compensated simulated annealing. In: Proceedings of SPIE, the International Society for Optical Engineering, vol. 1766 (1992)
5. Boden, B., Nnemann, S., Hoffmann, H., Seidl, T.: Mining coherent subgraphs in multi-layer graphs with edge labels. In: KDD, pp. 1258–1266 (2012)
6. Bogue, E.T., de Souza, C.C., Xavier, E.C., Freire, A.S.: An integer programming formulation for the maximum k-subset intersection problem. In: Lecture Notes in Computer Science, vol. 8596, pp. 87–99 (2014)
7. Chakraborty, T., Narayanam, R.: Cross-layer betweenness centrality in multiplex networks with applications. In: ICDE, pp. 397–408 (2016)
8. Chuang, J.R., Lin, J.M.: Efficient multi-layer obstacle-avoiding preferred direction rectilinear Steiner tree construction. In: Asia and South Pacific Design Automation Conference, pp. 527–532 (2011)
9. David, C.W.: Stirling's Approximation. Betascript Publishing, Saarbrücken (2007)
10. Dong, X., Frossard, P., Vandergheynst, P., Nefedov, N.: Clustering with multi-layer graphs: a spectral perspective. IEEE Trans. Signal Process. **60**(11), 5820–5831 (2011)
11. Fang, Y., Zhang, H., Ye, Y., Li, X.: Detecting hot topics from twitter: a multiview approach. J. Inf. Sci. **40**(5), 578–593 (2014)
12. Frickey, T., Weiller, G.: Mclip: motif detection based on cliques of gapped local profile-to-profile alignments. Bioinformatics **23**(4), 502–3 (2007)
13. Hu, H., Yan, X., Huang, Y., Han, J., Zhou, X.J.: Mining coherent dense subgraphs across massive biological networks for functional discovery. Bioinformatics **21**(suppl-1), i213 (2005)
14. Kim, J., Lee, J.G.: Community detection in multi-layer graphs: a survey. ACM SIGMOD Record **44**(3), 37–48 (2015)
15. Kivelä, M., Arenas, A., Barthelemy, M., Gleeson, J.P., Moreno, Y., Porter, M.A.: Multilayer networks. J. Complex Netw. **2**(3), 203–271 (2014)
16. Lee, V.E., Ruan, N., Jin, R., Aggarwal, C.C.: A survey of algorithms for dense subgraph discovery. In: Aggarwal, C.C., Wang, H. (eds.) Managing and Mining Graph Data, pp. 303–336. Springer, New York (2010)
17. Li, H., Nie, Z., Lee, W.C., Giles, L., Wen, J.R.: Scalable community discovery on textual data with relations. In: CIKM, pp. 1203–1212 (2008)
18. Li, R.H., Qin, L., Yu, J.X., Mao, R.: Influential community search in large networks. PVLDB **8**(5), 509–520 (2015)
19. Liu, J., Wang, C., Gao, J., Han, J.: Multi-view clustering via joint nonnegative matrix factorization. In: SDM, pp. 252–260 (2013)
20. Pei, J., Jiang, D., Zhang, A.: On mining cross-graph quasi-cliques. In: KDD, pp. 228–238 (2005)
21. Qi, G.J., Aggarwal, C.C., Huang, T.: Community detection with edge content in social media networks. In: ICDE, pp. 534–545 (2012)
22. Ruan, Y., Fuhry, D., Parthasarathy, S.: Efficient community detection in large networks using content and links. In: WWW, pp. 1089–1098 (2012)
23. Sanjeev, K., Gilbert, H.: Multilayer Networks. Wiley, Hoboken (2011)
24. Silva, A., Jr, W.M., Zaki, M.J.: Mining attribute-structure correlated patterns in large attributed graphs. PVLDB **5**(5), 466–477 (2012)
25. Solé-Ribalta, A., De Domenico, M., Gómez, S., Arenas, A.: Centrality rankings in multiplex networks. In: Proceedings of the 2014 ACM Conference on Web Science, pp. 149–155. ACM (2014)
26. Sun, Y., Yu, Y., Han, J.: Ranking-based clustering of heterogeneous information networks with star network schema. In: KDD, pp. 797–806 (2009)
27. Szklarczyk, D., Morris, J.H., Cook, H., Kuhn, M., Wyder, S., Simonovic, M., Santos, A., Doncheva, N.T., Roth, A., Bork, P.: The string database in 2017: quality-controlled protein-protein association networks, made broadly accessible. Nucleic Acids Res. **45**(Database), D362–D368 (2017)
28. Tang, W., Lu, Z., Dhillon, I.S.: Clustering with multiple graphs. In: ICDM, pp. 1016–1021 (2009)
29. Xu, Z., Ke, Y., Wang, Y., Cheng, H., Cheng, J.: A model-based approach to attributed graph clustering. In: SIGMOD, pp. 505–516 (2012)
30. Yan, X., Han, J.: gSpan: graph-based substructure pattern mining. In: ICDM, pp. 721–724 (2002)
31. Yang, Y., Yan, D., Wu, H., Cheng, J., Zhou, S., Lui, J.C.S.: Diversified temporal subgraph pattern mining. In: KDD, pp. 1965–1974 (2016)
32. Zeng, Z., Wang, J., Zhou, L., Karypis, G.: Coherent closed quasi-clique discovery from large dense graph databases. In: KDD, pp. 797–802 (2006)
33. Zhang, J., Kong, X., Yu, P.S.: Predicting social links for new users across aligned heterogeneous social networks. In: ICDM, pp. 1289–1294 (2013)
34. Zhou, P., Miao, G., Bing, B.: Cross-layer congestion control and scheduling in multi-hop OFDMA wireless networks. In: IEEE Conference on Global Telecommunications, pp. 1–6 (2009)
35. Zhou, Y., Cheng, H., Yu, J.X.: Graph clustering based on structural/attribute similarities. PVLDB **2**(1), 718–729 (2009)