The VLDB Journal (2024) 33:49–71
https://doi.org/10.1007/s00778-023-00797-x

REGULAR PAPER

A near-optimal approach to edge connectivity-based hierarchical graph decomposition

Lijun Chang[1] · **Zhiyi Wang**[1]

Received: 22 March 2022 / Revised: 24 March 2023 / Accepted: 14 April 2023 / Published online: 6 May 2023

Abstract
The problem of efficiently computing all k-edge-connected components (k-ECCs) of a graph G for *a user-given* k has been extensively studied recently in view of its importance in many applications. The k-ECCs of G for *all possible values of* k form a hierarchical structure; that is, any two different k-ECCs for the same k value are disjoint and any k-ECC is contained in a unique (k-1)-ECC. In this paper, we study the problem of efficiently constructing the hierarchy tree of the k-ECCs for all possible k values, for a graph G. The existing approaches TD and BU construct the hierarchy tree in either a top-down manner or a bottom-up manner, with both having the time complexity of $\mathcal{O}\big(\delta(G) \times \mathsf{T}_{\mathsf{KECC}}(G)\big)$, where $\delta(G)$ is the degeneracy of G and $\mathsf{T}_{\mathsf{KECC}}(G)$ is the time complexity of computing all k-ECCs of G for a specific k value. Here, the degeneracy of G is defined as the maximum value among the minimum vertex degrees of all subgraphs of G and is at most $\sqrt{m}$ where m is the number of edges in G. To improve the time complexity, we propose a divide-and-conquer approach DC running in $\mathcal{O}\big((\log \delta(G)) \times \mathsf{T}_{\mathsf{KECC}}(G)\big)$ time; this time complexity is optimal up to a logarithmic factor. However, a straightforward implementation of DC would take $\mathcal{O}((m+n)\log \delta(G))$ main-memory space, which could easily run out-of-memory when processing large graphs; here, n is the number of vertices in G. To reduce the main-memory footprint of our algorithm, we propose adjacency array-based techniques to optimize the space complexity to $2m + \mathcal{O}(n \log \delta(G))$ and denote our resulting algorithm by DC-AA. As a by-product of DC-AA, we also improve the space complexity of the state-of-the-art algorithm for computing all k-ECCs for a specific k to $2m + \mathcal{O}(n)$, by using the same technique as used in DC-AA. Finally, we propose optimization techniques to improve the practical efficiency of the existing approach BU and denote the space-optimized version of it as BU*-AA which runs in $\mathcal{O}\big(\delta(G) \times \mathsf{T}_{\mathsf{KECC}}(G)\big)$ time and $2m + \mathcal{O}(n)$ space. Extensive experiments on large real graphs and synthetic graphs demonstrate that our algorithms DC-AA and BU*-AA outperform the state-of-the-art approaches by up to 28 times in terms of running time and by up to 8 times in terms of main memory usage. In particular, our approach BU*-AA processes the Twitter graph, which has more than 1 billion undirected edges, in 29 min with 13.5 GB memory, while the state-of-the-art approaches take more than 13 h after our space optimization; note that the state-of-the-art approaches run out-of-memory if without our space optimization. Our empirical study also shows that BU*-AA, despite having a higher time complexity, performs better than DC-AA in practice. We also remark that BU*-AA is much simpler and easier to implement than DC-AA.

Keywords Edge connectivity · Hierarchical graph decomposition · Near optimal · Divide-and-conquer

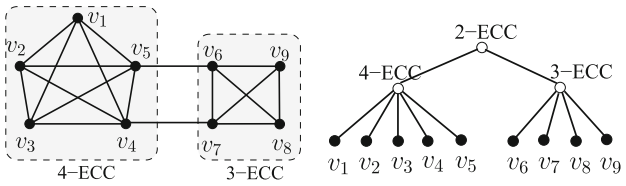
1 Introduction

The relationships among entities in real-world applications can be naturally modeled by the graph model. As a result, real graph data are abundant. With the proliferation of graph data, one of the fundamental graph problems is to compute the set of all *maximal* k-edge-connected subgraphs, called k-edge-connected components and abbreviated as k-ECCs, for a user-given k [3, 11, 40, 43]. A graph is *k-edge connected*, if it remains connected after removing any set of

✉ Lijun Chang
Lijun.Chang@sydney.edu.au

Zhiyi Wang
zwan9517@uni.sydney.edu.au

[1] School of Computer Science, The University of Sydney, Sydney, NSW, Australia



(a) A toy graph (b) ECo-decomposition

Fig. 1 A toy graph and its ECo-decomposition

$k-1$ edges. For the graph in Fig. 1a, the subgraphs induced by vertices $\{v_1, \ldots, v_5\}$ and $\{v_6, \ldots, v_9\}$ are two 3-ECCs, with the former also being a 4-ECC. Computing k-ECCs has many applications, such as identifying closely related entities for social behavior mining [2], discovering cohesive blocks (communities) in social networks (e.g., Facebook) [39], matrix completability analysis [14], and measuring robustness of communication networks [11].

Specifying the appropriate k value for an application is, however, non-trivial and usually needs several iterations of the trial-and-error process. In addition, different applications may need different k values. Hence, it is critical to precompute a data structure, such that all k-ECCs for any given k can be efficiently retrieved from the data structure which then as a result can facilitate downstream tasks. It is known that the k-ECCs for all possible k values form a hierarchical structure [42], as the different k-ECCs for the same k value are disjoint and each k-ECC is entirely contained in a unique $(k-1)$-ECC [7]. As an example, the *hierarchy tree* $\mathcal{T}$ for the k-ECCs of the graph G in Fig. 1a is shown in Fig. 1b. Here, the leaf nodes are vertices of G and non-leaf nodes correspond to k-ECCs of G; to distinguish vertices of $\mathcal{T}$ from that of G, we refer to vertices of $\mathcal{T}$ as nodes. From the constructed tree $\mathcal{T}$, the set of k-ECCs for any k can be easily extracted in time linear to the size of the k-ECCs. Thus, the problem becomes efficiently constructing the hierarchy tree for k-ECCs of all possible k values. We called this problem *Edge Connectivity-based hierarchical graph decomposition* and abbreviate it as ECo-*decomposition*.

Besides all the applications above, computing ECo-decomposition (i.e., the hierarchy tree) also has its own applications. Some example applications are as follows.

- *Hierarchical Organization and Visualization of Graphs.* ECo-decomposition constructs a hierarchical organization of a graph. This hierarchical organization can facilitate graph topology analysis [6] and assist users to visualize a graph in a multi-granularity manner [29], for example, zoom in and out based on the edge connectivities of subgraphs.
- *Graph Sparsification.* ECo-decomposition efficiently computes the Steiner connectivity for all edges (see Sect. 4.1). It is shown in [5, 20] that independently sampling edges according to their Steiner connectivities preserves the values of all cuts with a small multiplicative error. Note that this reduces the number of edges and thus sparsifies the graph.
- *Steiner Component Search.* ECo-decomposition is also an inherent preprocessing step for the problem of online Steiner component search [7, 25], which aims to compute the subgraph that contains a set of user-given query vertices and has the maximum edge connectivity [7].

The state-of-the-art approaches TD and BU compute the ECo-decomposition (i.e., construct the hierarchy tree $\mathcal{T}$) either in a top-down fashion [7] or in a bottom-up fashion [42]. TD computes k-ECCs of G for all possible k values in increasing order and thus constructs the hierarchy tree in a top-down fashion [7]. BU computes k-ECCs of G for all possible k values in decreasing order and thus constructs the hierarchy tree in a bottom-up fashion [42]. Computation sharing techniques are exploited in TD and BU based on the observation that the working graph in an iteration for computing k-ECCs could be smaller than the input graph G, e.g., the working graph in TD for computing k-ECCs is not G but the set of $(k-1)$-ECCs of G which are the results of the previous iteration [7]. The worst-case time complexities of TD and BU are both $\mathcal{O}\big(\delta(G) \times \mathsf{T}_{\mathsf{KECC}}(G)\big)$, where $\mathsf{T}_{\mathsf{KECC}}(G)$ is the time complexity of computing all k-ECCs of G for a specific k and $\delta(G)$ is the *degeneracy* of G. The degeneracy of G is defined as the maximum value among the minimum vertex degrees of all subgraphs of G [28]. Note that the time complexity of $\mathcal{O}\big(\delta(G) \times \mathsf{T}_{\mathsf{KECC}}(G)\big)$ is the same as the straightforward approach that *independently* computes k-ECCs of G for all possible k values, as the largest k will be no larger than $\delta(G)$.

Our Near-Optimal Approach. To improve the time complexity, we separate the computation into two parts, by first computing the Steiner connectivity for all edges of G and then constructing the hierarchy tree $\mathcal{T}$ based on the computed Steiner connectivities. Here, the *Steiner connectivity* of an edge (u, v), denoted $sc(u, v)$, is the largest k such that a k-ECC of G contains the edge. We in this paper show that the hierarchy tree of the ECo-decomposition can be constructed in $\mathcal{O}(m)$ time given the Steiner connectivities of all edges of G, where m is the number of edges of G. Consequently, the main problem of ECo-decomposition is to efficiently compute the Steiner connectivity for all edges of G.

We propose a divide-and-conquer approach DC to compute the Steiner connectivities of all edges. The general idea is that given the set E_L^H of edges of G whose Steiner connectivities are in the range $[L, H]$, i.e., $E_L^H = \{(u, v) \in E(G) \mid L \leq sc(u, v) \leq H\}$, we compute the exact Steiner connectivity for all edges of E_L^H as follows. If $L = H$, then $sc(u, v) = L$ for every edge $(u, v) \in E_L^H$ and the problem is solved. Otherwise, let $M = \lceil \frac{L+H}{2} \rceil$, we divide the problem into

two subproblems, E' and E'', to be solved recursively; here, $E' = E_L^{M-1} = \{(u, v) \in E(G) \mid L \leq sc(u, v) \leq M - 1\}$ and $E'' = E_M^H = \{(u, v) \in E(G) \mid M \leq sc(u, v) \leq H\}$. The critical step is efficiently dividing a search problem E_L^H into two subproblems: E' and E''. We show that E' is exactly the set of edges of E_L^H that are not in M-ECCs of the subgraph of G induced by E_L^H and all edges of G whose Steiner connectivities are larger than H, and E'' is equal to $E_L^H \backslash E'$. In addition, we propose computation sharing techniques and prove that DC runs in $\mathcal{O}\big((\log \delta(G)) \times \mathsf{T}_{\mathsf{KECC}}(G)\big)$ time.

The time complexity of DC is optimal up to a logarithmic factor, since the time complexity of any ECO-decomposition algorithm is clearly lower bounded by $\mathsf{T}_{\mathsf{KECC}}(G)$. However, a naive implementation of DC would take $\mathcal{O}\big((n+m) \log \delta(G)\big)$ main-memory space which makes it infeasible to process large graphs. We show that the space complexity can be improved to $\mathcal{O}(m + n \log \delta(G))$. However, this is still too high to be applied to billion-scale graphs as it will run out-of-memory. Note that the constant hidden by the big-$\mathcal{O}$ notation is large. In view of this, we further propose techniques, based on adjacency array, to reduce the space complexity to $2m + \mathcal{O}(n \log \delta(G))$ by explicitly bounding the constant on m by 2, while not increasing the time complexity. We denote our space-optimized approach as DC-AA, where AA stands for adjacency array.

In addition, we also propose a practically efficient algorithm BU*-AA, which follows the general paradigm of BU. That is, BU*-AA also computes k-ECCs of G for all possible values of k in decreasing order. However, instead of processing all vertices when computing the k-ECCs for each k, we show that we only need to process a subset of the vertices; we propose techniques to efficiently extract this subset of vertices as well as the subgraph induced by them. Although the time complexity of BU*-AA remains the same as that of BU, i.e., $\mathcal{O}(\delta(G) \times \mathsf{T}_{\mathsf{KECC}}(G))$, BU*-AA runs much faster than BU and performs even better than DC-AA in practice. Note that BU*-AA is much simpler and easier to implement than DC-AA. Furthermore, BU*-AA has a space complexity of $2m + \mathcal{O}(n)$.

We conduct extensive empirical studies on large graphs to evaluate our algorithms. The results show that our approach DC-AA outperforms the state-of-the-art approaches TD and BU by up to 28 times in terms of running time and by up to 8 times in terms of memory usage. Take the Twitter graph that has 1.2 billion undirected edges as an example, DC-AA finishes in 78 min by consuming 15 GB memory, while TD and BU (as well as DC) run out-of-memory on a machine with 128 GB memory; on the other hand, our space-optimized versions of TD and BU finish in 13.9 and 36.8 h, respectively. Furthermore, our practical approach BU*-AA runs faster and consumes less memory than DC-AA, e.g., BU*-AA finishes in 29 min with 13.5 GB memory for the Twitter graph.

Contributions. Our main contributions are summarized as follows:

1. We propose a near-optimal approach to ECO-decomposition, which improves the time complexity from $\mathcal{O}(\delta(G) \times \mathsf{T}_{\mathsf{KECC}}(G))$ of existing approaches to $\mathcal{O}((\log \delta(G)) \times \mathsf{T}_{\mathsf{KECC}}(G))$ of our approaches.
2. We propose techniques to implement our algorithm such that it takes $2m + \mathcal{O}(n \log \delta(G))$ main-memory space and thus can process billion-scale graphs in the main memory of a commodity machine.
3. We also propose a simpler and practically efficient algorithm that has space complexity of $2m + \mathcal{O}(n)$.
4. As a by-product of our space optimization techniques, we significantly reduce the memory usage of the state-of-the-art k-ECC computation algorithm proposed in [11]. Moreover, we anticipate that our space optimization techniques can also be generally applied to other graph algorithms.
5. We conduct extensive empirical studies on large real and synthetic graphs to evaluate the efficiency of our approaches.

A preliminary version of this work has been published in [10]. Compared with [10], we have proposed a new algorithm BU*-AA, added proofs for all lemmas and theorems, and conducted new experimental studies on SSCA synthetic graphs as well as the new algorithm BU*-AA in this paper.

Organization. The rest of the paper is organized as follows. Section 2 gives preliminaries of the studied problem, and Sect. 3 presents the existing algorithms. We propose a near-optimal approach in Sect. 4 and develop techniques to reduce the memory usage of our algorithms in Sect. 5. We further propose a simpler and practically efficient algorithm in Sect. 6. Section 7 reports the results of our experimental studies, and Sect. 8 provides an overview of related works. Finally, Sect. 9 concludes the paper.

2 Preliminaries

In this paper, we consider a large *unweighted and undirected graph* $G = (V, E)$, with vertex set V and edge set E. The number of vertices and the number of *undirected* edges in G are denoted by $n = |V|$ and $m = |E|$, respectively. Given a vertex subset $V_s \subseteq V$, the subgraph of G induced by vertices V_s is denoted by $G[V_s] = (V_s, \{(u, v) \in E \mid u, v \in V_s\})$. Given an edge subset $E_s \subseteq E$, the subgraph of G induced by edges E_s is denoted by $G[E_s] = (\cup_{(u,v)\in E_s}\{u, v\}, E_s)$. For an arbitrary graph g, we use $V(g)$ and $E(g)$ to, respectively, denote its set of vertices and its set of edges.

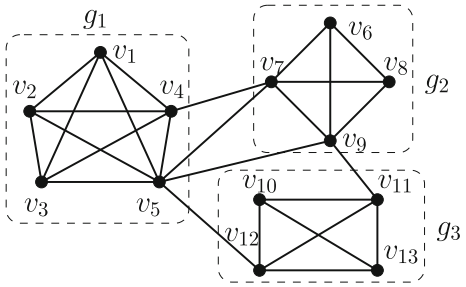


Fig. 2 An example graph

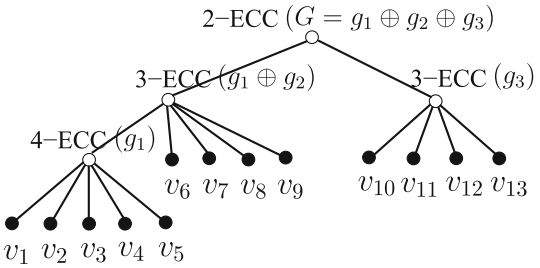


Fig. 3 Hierarchy tree $\mathcal{T}$

A graph is *k-edge connected* if the remaining graph is still connected after the removal of any $k-1$ edges from it. Note that, by definition, a graph with less than k edges (e.g., consisting of a singleton vertex) is not considered to be k-edge connected. Then, k-edge-connected component is defined as follows.

Definition 1 (*k-edge-Connected Component* [11]) Given a graph G, a subgraph g of G is a k-edge-connected component (abbreviated as k-ECC) of G if (i) g is k-edge connected and (ii) g is *maximal* (i.e., any super-graph of g is not k-edge connected).

Consider the graph in Fig. 2, the entire graph is a 2-ECC but not a 3-ECC (since the graph will be disconnected after removing edges (v_5, v_{12}) and (v_9, v_{11})). The subgraph g_1 is a 4-ECC, and g_3 is a 3-ECC. Note that g_2, although is 3-edge connected, is not a 3-ECC since its super-graph $g_1 \oplus g_2$ is also 3-edge connected (i.e., g_2 is not maximal). Here, $g_1 \oplus g_2$ denotes the union of g_1 and g_2, which also includes the cross edges between vertices of g_1 and vertices of g_2.

Hierarchy Tree of k-ECCs. It is shown in [7] that the k-ECCs of a graph satisfy the following properties.

1. Each k-ECC is a vertex-induced subgraph.
2. Any two distinct k-ECCs for the same k value are disjoint.
3. Each k-ECC for $k > 1$ is entirely contained in a $(k-1)$-ECC.

Thus, the k-ECCs of a graph G for all possible k values can be compactly represented by a *hierarchy tree* $\mathcal{T}$, where leaf nodes of $\mathcal{T}$ correspond to vertices of G and non-leaf nodes of $\mathcal{T}$ correspond to distinct k-ECCs of G. Note that, to distinguish vertices of $\mathcal{T}$ from that of G, we refer to vertices of $\mathcal{T}$ as *nodes*. Figure 3 illustrates the hierarchy tree for k-ECCs of the graph in Fig. 2.

We call non-leaf nodes of $\mathcal{T}$ as ECC nodes, and each ECC node is associated with a weight. An ECC node of weight k corresponds to a k-ECC which is the subgraph of G induced by all leaf nodes in the subtree of $\mathcal{T}$ rooted at the ECC node. For example, the left 3-ECC node in Fig. 3 corresponds to the 3-ECC $g_1 \oplus g_2$ in Fig. 2, which is the subgraph induced by vertices $v_1, \ldots, v_9$. Note that if a subgraph g is both a k-ECC and a $(k+1)$-ECC, it is only represented once in the hierarchy tree by an ECC node of weight $k+1$. For example, the entire graph G is both a 2-ECC and a 1-ECC and is represented by the ECC node of weight 2. Thus, each non-leaf node will have at least two children, and *the size of the hierarchy tree* $\mathcal{T}$ *is linear to* n.

It is worth pointing out that for any given k, the set of all k-ECCs of G can be efficiently obtained from the hierarchy tree $\mathcal{T}$ in time linear to the size of the k-ECCs.

Problem Statement. Given a large graph G, we study the problem of efficiently constructing the hierarchy tree for the set of all k-ECCs of G. We term this problem as *Edge Connectivity-based hierarchical graph decomposition* and abbreviate it as ECO-*decomposition*.

In this paper, we will consider the algorithm for computing all k-ECCs of g for a given k as a black-box, denoted $\mathsf{KECC}(g, k)$. While any of the algorithms in [3, 11, 43] can be used to implement $\mathsf{KECC}(g, k)$, we implement the state-of-the-art algorithm in [11] in our experiments. We use $\mathsf{T}_{\mathsf{KECC}}(G)$ to denote the time complexity of $\mathsf{KECC}(G, k)$ when G is taken as the input graph[1] and assume that $\mathsf{T}_{\mathsf{KECC}}(G)$ is at least linear to the number of edges of G.

3 Existing solutions

In this section, we briefly review the two state-of-the-art approaches, and discuss their time complexities. The existing approaches compute the ECO-decomposition (i.e., the hierarchy tree) either in a top-down manner [7] or in a bottom-up manner [42].

A Top-Down Approach: TD. The top-down approach constructs the hierarchy tree in a top-down manner, which is achieved by explicitly computing k-ECCs of G for all k values in increasing order [7]. The pseudocode is shown in

[1] Note that a more rigid definition of $\mathsf{T}_{\mathsf{KECC}}(G)$ would be $\max_k \mathsf{T}_{\mathsf{KECC}}(G, k)$, but all existing time complexity analyses of $\mathsf{KECC}(G, k)$ are independent of k.

Algorithm 1: TD(G)

```
1  Create the root ECC node r of T with weight 1;
2  Construct-TD(r, 1, G);
3  return T;

   Procedure Construct-TD(ecc, k, g)
4  φ_{k+1}(g) ← KECC(g, k + 1);
5  if φ_{k+1}(g) is the same as g (i.e., g ∈ φ_{k+1}(g)) then
6      Change the weight of ecc to k + 1;
7      Construct-TD(ecc, k + 1, g);
8  else
9      foreach vertex v of g that is not in subgraphs of φ_{k+1}(g) do
10         Create a leaf node for v to be a child of ecc in T;
11     foreach connected subgraph g' ∈ φ_{k+1}(g) do
12         Create an ECC node ecc' of weight k + 1 to be a child of
           ecc in T;
13         Construct-TD(ecc', k + 1, g');
```

Algorithm 1, denoted by TD. Initially, the root ECC node r of weight 1, which corresponds to the entire input graph G, is created for $\mathcal{T}$ (Line 1); note that, without loss of generality here G is assumed to be connected. Then, it recursively adds the set of children to each ECC node in a top-down fashion by invoking CONSTRUCT- TD (Line 2).

Given an ECC node ecc of weight k whose corresponding graph is g (i.e., g is a k-ECC of G), CONSTRUCT- TD constructs the set of children of ecc. To do so, it first computes the set of $(k+1)$-ECCs of g (Line 4), denoted $\phi_{k+1}(g)$. If $\phi_{k+1}(g)$ is the same as g which means that g itself is $(k+1)$-edge connected (Line 5), then the weight of ecc is increased to $k+1$ (Line 6) and the recursion continues for g (Line 7). Otherwise, the set of children of ecc is added as follows: (i) a leaf node is added for each vertex of g that is not in $\phi_{k+1}(g)$ (Lines 9–10) and (ii) an ECC node is added for each connected subgraph g' of $\phi_{k+1}(g)$ (Lines 11–12). The recursion continues for each newly added ECC node (Line 13).

Example 1 Consider the graph in Fig. 2. $\phi_2(G) = \{G\}$; thus, the weight of the root node r of $\mathcal{T}$ is 2. $\phi_3(G) = \{g_1 \oplus g_2, g_3\}$; thus, r has two children corresponding to $g_1 \oplus g_2$ and g_3, respectively. $\phi_4(g_3) = \emptyset$; thus, all vertices of g_3 are added to be the children of the ECC node corresponding to g_3. The final hierarchy tree $\mathcal{T}$ is the same as shown in Fig. 3.

A Bottom-Up Approach: BU. The bottom-up approach constructs the hierarchy tree in a bottom-up fashion, which is achieved by computing k-ECCs of G for all k values in decreasing order [42]. The pseudocode is shown in Algorithm 2, denoted BU. Initially, one leaf node is created in $\mathcal{T}$ for each vertex of G (Line 1), and an upper bound $\overline{k_{\max}}(G)$ of the largest k such that G has a non-empty k-ECC is computed (Line 2). Then, the set of k-ECCs $\phi_k(G)$ of G are computed, for k varying from $\overline{k_{\max}}(G)$ to 1 (Lines 3–4).

Algorithm 2: BU(G)

```
1  Create one leaf node in T for each vertex of G;
2  Compute an upper bound k̄_max(G) of the largest k such that G
   has a non-empty k-ECC;
3  for k ← k̄_max(G) down to 1 do
4      φ_k(G) ← KECC(G, k);
5      foreach connected subgraph g ∈ φ_k(G) do
6          Create an ECC node ecc in T with weight k;
7          Add the set of nodes of T that correspond to vertices of g
           to be the children of ecc;
8          Contract g into a single super-vertex in G, to which ecc
           corresponds;
9  return T;
```

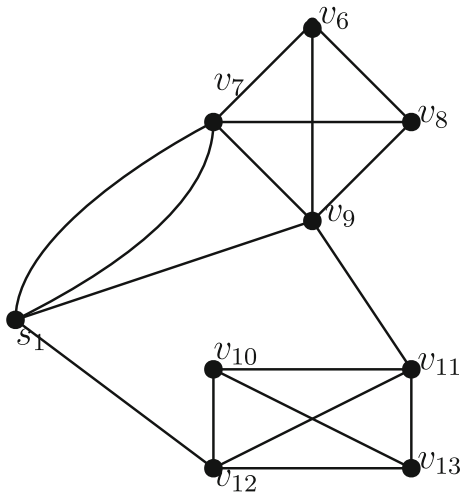


Fig. 4 The graph in Fig. 2 after contracting g_1 into super-vertex s_1

For each connected subgraph g in $\phi_k(G)$ (Line 5), an ECC node ecc is created in $\mathcal{T}$ (Line 6), its children are the nodes of $\mathcal{T}$ that correspond to vertices of G (Line 7), and g is then contracted into a super-vertex in G (Line 8); thus, ecc corresponds to the resulting super-vertex in G obtained by contracting g. Note that, when computing k-ECCs of G, each $(k+1)$-ECC of G has already been contracted into a super-vertex. As a result, if a subgraph is both a k-ECC and a $(k-1)$-ECC, it will only be computed, once, as a k-ECC. The correctness of BU follows from the fact that for a k-edge-connected subgraph g that is not $(k+1)$-edge connected, if we contract each $(k+1)$-ECC of g into a super-vertex, then the resulting graph is still k-edge connected.

Example 2 Consider the graph in Fig. 2 and assume the upper bound is computed as $\overline{k_{\max}}(G) = 4$. In the first iteration, we obtain $\phi_4(G) = \{g_1\}$; thus, an ECC node of weight 4 is created in $\mathcal{T}$ with five children v_1, v_2, v_3, v_4, v_5. We contract g_1 into a single super-vertex, denoted s_1; the resulting graph is shown in Fig. 4 with two parallel edges between s_1 and v_7. In the second iteration, we obtain $\phi_3(G) = \{s_1 \oplus g_2, g_3\}$; thus, two ECC nodes of weight 3 are created in $\mathcal{T}$, corresponding to

these two subgraphs, respectively. Then, $s_1 \oplus g_2$ is contracted into a super-vertex, denoted $s_{1,2}$, and g_3 is contracted into a super-vertex, denoted s_3. In the third iteration, we obtain $\phi_2(G) = \{s_{1,2} \oplus s_3\}$; thus, the root node of $\mathcal{T}$ is obtained. The final result is shown in Fig. 3.

Time Complexities of TD and BU. We first prove the following lemma.

Lemma 1 *Let $k_{\max}(G)$ be the largest k such that G contains a non-empty k-ECC and $\delta(G)$ be the degeneracy of G which is equal to the maximum value among the minimum vertex degrees of all subgraphs of G [28]. Then, we have $k_{\max}(G) \leq \delta(G)$.*

Proof We prove the lemma by contradiction. Suppose $k_{\max}(G) > \delta(G)$, and let g be a $(k_{\max}(G))$-ECC of G. Then, the minimum vertex degree of g is at least $k_{\max}(G)$, which is larger than $\delta(G)$. This contradicts the fact that $\delta(G)$ equals the maximum value among the minimum vertex degree of all subgraphs of G. Thus, the lemma holds. □

We actually observe that $k_{\max}(G) = \delta(G)$ for all real and synthetic graphs tested in our experiments. Thus, the largest k that is input to CONSTRUCT- TD of Algorithm 1 is $\delta(G)$. As we assumed that $\mathsf{T}_{\mathsf{KECC}}(G)$ is at least linear to the number of edges of G, the time complexity of TD is $\mathcal{O}\big(\delta(G) \times \mathsf{T}_{\mathsf{KECC}}(G)\big)$.[2] Note that the time complexity analysis of TD is tight; for example, consider an input graph G that itself is $\delta(G)$-edge connected.

Following Lemma 1, the upper bound $\overline{k_{\max}}(G)$ can be set as $\delta(G)$ at Line 2 of Algorithm 2. Thus, the time complexity of BU is $\mathcal{O}\big(\delta(G) \times \mathsf{T}_{\mathsf{KECC}}(G)\big)$,[3] as the degeneracy of G can be computed in $\mathcal{O}(m)$ time [28]. Note that the time complexity analysis of BU is also tight; for example, consider a graph that has no k-ECCs other than a $\delta(G)$-ECC and G itself which is 2-edge connected.

The degeneracy $\delta(G)$, although can be bounded by $\mathcal{O}(\sqrt{m})$ in the worst case [36], may still be large, especially for large graphs. For example, $\delta(G)$ is more than 2000 for the largest graphs tested in our experiments (see Table 1 in Sect. 7). As a result, BU and TD are taking excessively long time for processing large graphs due to their high time complexity of $\mathcal{O}\big(\delta(G) \times \mathsf{T}_{\mathsf{KECC}}(G)\big)$, not to mention their high space complexity (see Sect. 5).

[2] Although the time complexity of TD is analyzed to be $\mathcal{O}\big(\alpha(G) \times \mathsf{T}_{\mathsf{KECC}}(G)\big)$ in [7] where $\alpha(G)$ is the arboricity of G, this is the same as $\mathcal{O}\big(\delta(G) \times \mathsf{T}_{\mathsf{KECC}}(G)\big)$ since $\alpha(G) \leq \delta(G) \leq 2\alpha(G) - 1$ [36].

[3] It is worth pointing out that the original algorithm in [42] is designed for I/O-efficient settings, and its time complexity cannot be bounded by $\mathcal{O}\big(\delta(G) \times \mathsf{T}_{\mathsf{KECC}}(G)\big)$ as the upper bound $\overline{k_{\max}}(G)$ is set as the maximum degree of G in [42].

Handling Dynamic Graphs. Techniques for handling dynamic graphs have also been proposed in [7]. The general idea is based on the fact that deleting an edge from a graph or inserting a new edge into a graph will change the edge connectivity of the graph by at most 1, and moreover, most of the k-ECCs will remain unchanged. These techniques can be directly adopted to maintain the hierarchy tree for dynamic graphs. We omit the details, as we focus on speeding up the construction of the hierarchy tree in this paper.

4 A near-optimal approach

In this section, we propose an approach for ECO-decomposition that runs in $\mathcal{O}\big((\log \delta(G)) \times \mathsf{T}_{\mathsf{KECC}}(G)\big)$ time. To achieve this, we will need to avoid the explicit computation and enumeration of k-ECCs for all possible k values, since the k-ECCs for all possible k values could be of total size $\delta(G) \times \mathsf{T}_{\mathsf{KECC}}(G)$ in the worst case if $\mathsf{T}_{\mathsf{KECC}}(G)$ is roughly linear to m. Consequently, simply reporting k-ECCs for all possible k values would already take $\mathcal{O}\big(\delta(G) \times \mathsf{T}_{\mathsf{KECC}}(G)\big)$ time. Instead, we use a two-step paradigm, which first computes the Steiner connectivity for all edges of G and then constructs the hierarchy tree based on the Steiner connectivities, as follows.

1 Step-I: Compute the steiner connectivity for all edges of G;
2 Step-II: Construct the hierarchy tree based on the computed steiner connectivities;

We will show that given the Steiner connectivities of all edges, the hierarchy tree can be constructed in linear time, and thus, computing the Steiner connectivities of all edges dominates the time complexity. As there are only m edges and thus only m Steiner connectivities to compute, it is possible to compute them in time less than $\delta(G) \times \mathsf{T}_{\mathsf{KECC}}(G)$.

In the following, we first in Sect. 4.1 propose an algorithm to compute the Steiner connectivities of all edges in $\mathcal{O}\big((\log \delta(G)) \times \mathsf{T}_{\mathsf{KECC}}(G)\big)$ time and then in Sect. 4.2 present an algorithm to construct the hierarchy tree in $\mathcal{O}(m)$ time based on the computed Steiner connectivities.

4.1 Computing Steiner connectivities

The Steiner connectivity is defined as follows.

Definition 2 **(Steiner Connectivity** [7]) Given a graph G, the Steiner connectivity of an edge (u, v), denoted $sc(u, v)$, is the largest k such that a k-ECC of G contains both u and v.

For example, in Fig. 5, the Steiner connectivity of each edge is computed as shown on the edge, e.g., $sc(v_1, v_4) = 4$.

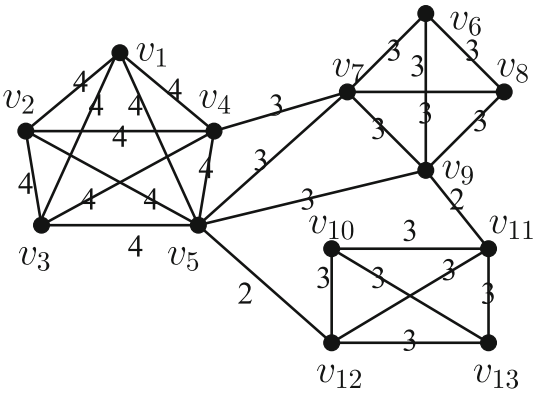


Fig. 5 Steiner connectivities

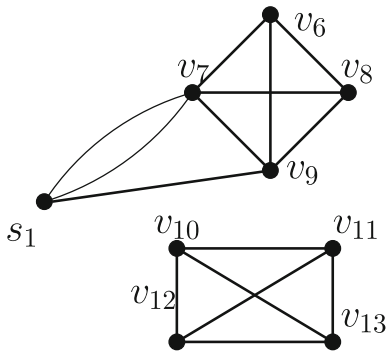


Fig. 6 $\mathtt{GS}_3^3(G)$ for the graph in Fig. 2

Given a graph G, let $\phi_k(G)$ be the set of k-ECCs of G, then all edges of $\phi_k(G)$ have Steiner connectivity at least k and all edges of G that are not in $\phi_k(G)$ have Steiner connectivity smaller than k. In this subsection, we propose a divide-and-conquer approach for computing the Steiner connectivities of all edges in a graph. Note that, although the concept of Steiner connectivity is borrowed from [7], all our techniques in the following are new.

A Graph Shrink Operator $\mathtt{GS}_{k_1}^{k_2}(\cdot)$. We first introduce a graph shrink operator $\mathtt{GS}_{k_1}^{k_2}(\cdot)$ for $k_1 \le k_2$. Given a graph G, the result of $\mathtt{GS}_{k_1}^{k_2}(G)$ is still a graph. It is obtained from G by (1) removing all vertices and edges that are not in k_1-ECCs of G and (2) contracting each (k_2+1)-ECC of G into a super-vertex. Note that the resulting graph of $\mathtt{GS}_{k_1}^{k_2}(\cdot)$ may have parallel edges. For example, $\mathtt{GS}_3^3(G)$ for the graph G in Fig. 2 is shown in Fig. 6 which is obtained by (1) removing edges (v_5, v_{12}) and (v_9, v_{11}) and (2) contracting subgraph g_1 into a super-vertex s_1. There are two parallel edges between s_1 and v_7 in Fig. 6.

The graph shrink operator $\mathtt{GS}_{k_1}^{k_2}(\cdot)$ has several properties which will be useful for computing Steiner connectivities. Firstly, applying the operator $\mathtt{GS}_{k_1}^{k_2}(\cdot)$ preserves the Steiner connectivity for all edges in the resulting graph.

Property 1 *Given a graph G and two integers $k_1 \le k_2$, the Steiner connectivity of each edge of $\mathtt{GS}_{k_1}^{k_2}(G)$ when computed in $\mathtt{GS}_{k_1}^{k_2}(G)$ is the same as that computed in G.*

Proof Consider an arbitrary edge (u, v) in $\mathtt{GS}_{k_1}^{k_2}(G)$. It is easy to verify that the Steiner connectivity of (u, v) must be within the range $[k_1, k_2]$ when computed in $\mathtt{GS}_{k_1}^{k_2}(G)$ and also when computed in G. Assume the Steiner connectivity of (u, v) in G is $k \in [k_1, k_2]$, and let g be the k-ECC of G that contains (u, v). Then, by contracting each (k_2+1)-ECC of g into a super-vertex, the resulting graph is still k-edge connected and is a k-ECC of $\mathtt{GS}_{k_1}^{k_2}(G)$. Thus, the Steiner connectivity of (u, v) when computed in $\mathtt{GS}_{k_1}^{k_2}(G)$ is no smaller than that computed in G. Similarly, we can prove that the Steiner connectivity of (u, v) when computed in G is no smaller than that computed in $\mathtt{GS}_{k_1}^{k_2}(G)$. Therefore, the property holds. □

Secondly, the Steiner connectivity for all edges of $\mathtt{GS}_k^k(G)$ is k. For example, all edges in Fig. 6 have Steiner connectivity 3.

Property 2 *Given a graph G and an integer k, every edge of $\mathtt{GS}_k^k(G)$ has Steiner connectivity k.*

Proof This property directly follows from Property 1 and the definition of Steiner connectivity. □

Thirdly, multiple operations of $\mathtt{GS}_{k_1}^{k_2}(\cdot)$ can be chained together.

Property 3 *Given a graph G and four integers $k_1 \le k_2$ and $k_3 \le k_4$ such that $\max\{k_1, k_3\} \le \min\{k_2, k_4\}$, we have $\mathtt{GS}_{k_3}^{k_4}\big(\mathtt{GS}_{k_1}^{k_2}(G)\big) = \mathtt{GS}_{\max\{k_1,k_3\}}^{\min\{k_2,k_4\}}(G)$.*

Proof This property directly follows from properties 1 and 2 and the hierarchical structure of k-ECCs for different k values. □

Our Divide-and-Conquer Approach: DC. From Property 2, we know that the Steiner connectivities of all edges of $\mathtt{GS}_k^k(G)$ are k. Moreover, from the definitions of Steiner connectivity and the graph shrink operator, we know that all edges whose Steiner connectivities are k will be in $\mathtt{GS}_k^k(G)$. Thus, to compute Steiner connectivities of all edges of G, it suffices to compute $\mathtt{GS}_k^k(G)$ for $k \in [1, \delta(G)]$. Instead of naively computing $\mathtt{GS}_k^k(G)$ independently for each $k \in [1, \delta(G)]$ which would take $\mathcal{O}\big(\delta(G) \times \mathsf{T}_{\mathsf{KECC}}(G)\big)$ time, we propose a divide-and-conquer approach based on the fact that $\mathtt{GS}_k^k(G)$ is entirely contained in $\mathtt{GS}_{k_1}^{k_2}$ if $k_1 \le k \le k_2$.

The pseudocode of our approach is shown in Algorithm 3, denoted DC. It first computes the degeneracy $\delta(G)$ of G (Line 1) and then invokes procedure Compute-DC with input $(G, 1, \delta(G))$ to compute the Steiner connectivities of all edges (Line 2), while Line 3 constructs the hierarchy tree

Algorithm 3: DC(G)

```
1  Compute the degeneracy δ(G) of G;
2  Compute-DC(G, 1, δ(G));
3  ConstructHierarchy(G, sc(·,·));      /* See Algorithm 4 */;
4  return T;

   Procedure Compute-DC(g, L, H)
5  if L = H then
6      foreach edge (u, v) ∈ E(g) do sc(u, v) ← L;
7  else
8      Choose an integer M such that L < M ≤ H;
9      φ_M(g) ← KECC(g, M);      /* Compute M-ECCs of g */;
10     Let g_1 be the graph obtained from g by contracting each
       connected subgraph of φ_M(g) into a super-vertex, and g_2 be
       the union of subgraphs in φ_M(g);      /* g_1 = GS_L^{M-1}(G),
       g_2 = GS_M^H(G) */;
11     Compute-DC(g_1, L, M − 1);
12     Compute-DC(g_2, M, H);
```

and will be discussed in Sect. 4.2. The input to Compute-DC consists of a graph g and an interval $[L, H]$. If $L = H$, then the Steiner connectivities of all edges of g are set as L (Lines 5–6). Otherwise, an integer M is chosen such that $L < M \leq H$ (Line 8), then the set $\phi_M(g)$ of M-ECCs of g is computed (Line 9) and two graphs g_1 and g_2 are obtained from g based on $\phi_M(g)$ (Line 10), and finally the algorithm continues on g_1 (Line 11) and on g_2 (Line 12).

We prove by the following lemma that when initially invoking Compute-DC with graph G and interval $[1, \delta(G)]$, the graph g being processed for each recursion with interval $[L, H]$ is $\mathtt{GS}_L^H(G)$.

Lemma 2 *For* Compute-DC*, if the input graph g is $\mathtt{GS}_L^H(G)$, then the two graphs g_1 and g_2 obtained at Line 10 are exactly $\mathtt{GS}_L^{M-1}(G)$ and $\mathtt{GS}_M^H(G)$, respectively.*

Proof According to the definition of the graph shrink operation $\mathtt{GS}_{k_1}^{k_2}(\cdot)$, it is easy to see that $g_1 = \mathtt{GS}_L^{M-1}(g)$ and $g_2 = \mathtt{GS}_M^H(g)$. Based on the assumption that $g = \mathtt{GS}_L^H(G)$ and $L < M \leq H$ (see Line 8 of Algorithm 3), we have $g_1 = \mathtt{GS}_L^{M-1}(g) = \mathtt{GS}_L^{M-1}(\mathtt{GS}_L^H(G)) = \mathtt{GS}_L^{M-1}(G)$ where the last equality follows from Property 3. Similarly, we have $g_2 = \mathtt{GS}_M^H(g) = \mathtt{GS}_M^H(\mathtt{GS}_L^H(G)) = \mathtt{GS}_M^H(G)$. Thus, the lemma holds. □

Based on Lemma 2, the recursion tree of invoking Compute-DC with input $(G, 1, \delta(G))$ is shown in Fig. 7.

The correctness and time complexity of Algorithm 3 are proven by the following two theorems.

Theorem 1 *Algorithm 3 correctly computes the Steiner connectivity for all edges of G.*

Proof The recursive invocation of Compute-DC stops when $L = H$ (see Lines 5–6 of Algorithm 3). If $L < H$, then Compute-DC recursively solves two instances, one for $k \in [L, M-1]$ (Line 11), and another for $k \in [M, H]$. Thus, it is easy to see that for each $k \in [1, k_{\max}(G)]$, there is an instance of Compute-DC with input $L = H = k$. Moreover, according to Lemma 2, the input graph to this instance is $g = \mathtt{GS}_k^k(G)$, as the input graph to the first invocation of Compute-DC is $\mathtt{GS}_1^{\delta(G)}(G)$ which is the same as G. Thus, the recursion tree of invoking Compute-DC with input $(G, 1, \delta(G))$ is shown in Fig. 7. Property 2 states that the instance of GS with input $g = \mathtt{GS}_k^k(G)$, $L = k$ and $H = k$ correctly computes the Steiner connectivity for all edges whose Steiner connectivities are k. Thus, the theorem holds. □

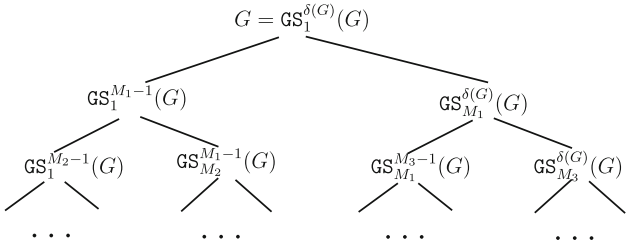


Fig. 7 Recursion tree

Theorem 2 *The time complexity of Algorithm 3 is $\mathcal{O}(h \times \mathsf{T}_{\mathsf{KECC}}(G))$, where h is the height of the recursion tree in Fig. 7.*

Proof We define the level of a node in the recursion tree as its distance to the root. Then, the subgraphs of G, corresponding to the nodes of the recursion tree that are at the same level, have disjoint sets of edges. Thus, the time complexity of the computation for each level of the recursion tree is $\mathcal{O}(\mathsf{T}_{\mathsf{KECC}}(G))$, by assuming that $\mathsf{T}_{\mathsf{KECC}}(G)$ is linear or superliner to the number of edges of G. Consequently, the total time complexity of Algorithm 3 is $\mathcal{O}(h \times \mathsf{T}_{\mathsf{KECC}}(G))$ as the recursion tree has h levels. □

Near-Optimal Time Complexity. Algorithm 3 correctly computes the Steiner connectivities of all edges regardless of the choice of M at Line 8, as long as $L < M \leq H$. Yet, the time complexity of Algorithm 3 would vary for different choices of M. For example, if M is always set as $L + 1$ or always set as H, then the height of the recursion tree would be $\delta(G)$ and thus the time complexity of Algorithm 3 would be $\mathcal{O}(\delta(G) \times \mathsf{T}_{\mathsf{KECC}}(G))$ on the basis of Theorem 2. To make the time complexity as low as possible, we will need to reduce the height of the recursion tree. Thus, we propose to set M as $\lceil \frac{L+H}{2} \rceil$ and prove in the following theorem that the time complexity of Algorithm 3 then becomes $\mathcal{O}\big((\log \delta(G)) \times \mathsf{T}_{\mathsf{KECC}}(G)\big)$.

Theorem 3 *By setting $M = \lceil \frac{L+H}{2} \rceil$, the time complexity of Algorithm 3 is $\mathcal{O}\big((\log \delta(G)) \times \mathsf{T}_{\mathsf{KECC}}(G)\big)$.*

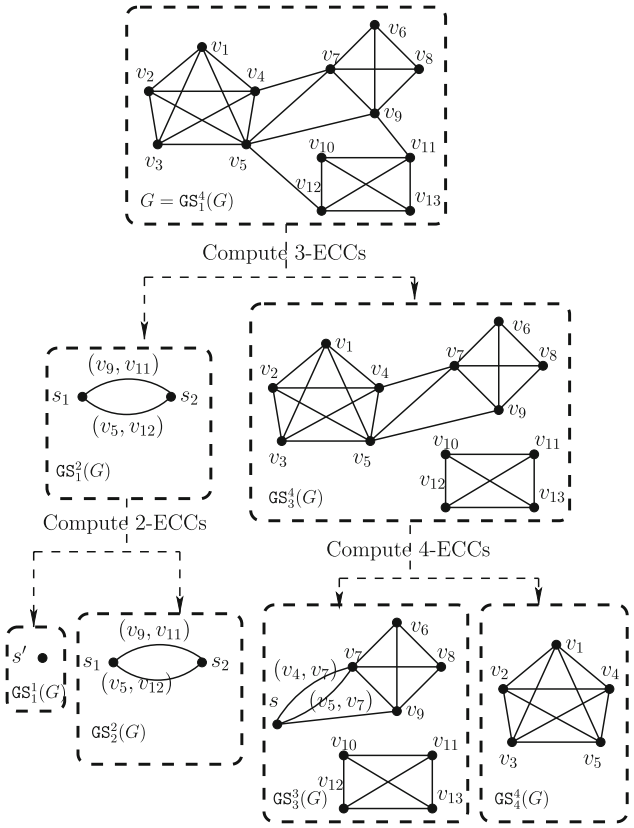


Fig. 8 Running example of DC

Proof By setting $M = \lceil \frac{L+H}{2} \rceil$, the length of the interval $[L, H]$ for each node in the recursion tree will be only half of that of its parent. Thus, the height of the recursion tree becomes $\mathcal{O}(\log \delta(G))$, as the interval for the root node is $[1, \delta(G)]$. Consequently, the time complexity of Algorithm 3 is $\mathcal{O}\big((\log \delta(G)) \times \mathsf{T}_{\mathsf{KECC}}(G)\big)$ by following Theorem 2. □

Following the above theorem, we set $M = \lceil \frac{L+H}{2} \rceil$ in Algorithm 3. The time complexity of DC, which is $\mathcal{O}\big((\log \delta(G)) \times \mathsf{T}_{\mathsf{KECC}}(G)\big)$, is optimal up to a logarithmic factor $\log \delta(G)$. This is because the time complexity of ECO-decomposition cannot be lower than $\mathsf{T}_{\mathsf{KECC}}(G)$, as ECO-decomposition also implicitly computes the k-ECCs of G; specifically, the k-ECCs of G can be obtained from the hierarchy tree in time linear to the sizes of the k-ECCs.

Example 3 Here, we apply DC on the graph G in Fig. 2 as an example. Figure 8 indicates the whole running process of DC on G, where the top-most part is G itself. The degeneracy is $\delta(G) = 4$. Then, we compute the Steiner connectivities of all edges of G by invoking Compute-DC with input G and $[L, H] = [1, 4]$. Here, $\mathtt{GS}_1^4(G)$ is the same as G. As $L \neq H$ and $\lceil \frac{L+H}{2} \rceil = 3$, we compute the 3-ECCs of G and obtain the subgraphs induced by $S_1 = \{v_1, v_2, \ldots, v_9\}$ and $S_2 = \{v_{10}, \ldots, v_{13}\}$, respectively. Thus, we obtain the two graphs $\mathtt{GS}_1^2(G)$ and $\mathtt{GS}_3^4(G)$ as shown in the middle layer of Fig. 8. The computation continues on these two graphs with intervals $[1, 2]$ and $[3, 4]$, respectively.

The graph $\mathtt{GS}_3^4(G)$ is composed of the two 3-ECCs of G as shown in right part of the middle layer of Fig. 8. We compute the 4-ECCs of $\mathtt{GS}_3^4(G)$ and obtain the subgraph induced by vertices $\{v_1, v_2, \ldots, v_5\}$. Thus, all edges among vertices $\{v_1, v_2, \ldots, v_5\}$ have Steiner connectivities 4 as indicated in $\mathtt{GS}_4^4(G)$, while the other edges have Steiner connectivities 3 as demonstrated in $\mathtt{GS}_3^3(G)$.

The graph $\mathtt{GS}_1^2(G)$ is obtained by contracting each of S_1 and S_2 into a super-vertex as shown in the left part of the middle layer of Fig. 8. In $\mathtt{GS}_1^2(G)$, there are two parallel edges between s_1 and s_2, corresponding to edges (v_9, v_{11}) and (v_5, v_{12}). As $\mathtt{GS}_1^2(G)$ is 2-edge connected, the Steiner-connectivities of (v_9, v_{11}) and (v_5, v_{12}) are 2. □

Algorithm 4: ConstructHierarchy

Input: A graph G with $sc(u, v)$ for each edge (u, v)
Output: The hierarchy tree of ECO-decomposition of G

```
1  Initialize an empty hierarchy tree T;
2  foreach vertex u ∈ V(G) do Insert a singleton node u into T;
3  foreach edge (u, v) ∈ E(G) in non-increasing sc(u, v) order do
4      Let r_u (resp r_v) be the root of the tree in T containing u (resp v);
5      if r_u = r_v then continue;
6      else if both r_u and r_v are ECC nodes with weight sc(u, v) then
7          Merge r_u and r_v into a single ECC node;
8      else if none of r_u or r_v is an ECC node with weight sc(u, v) then
9          Create a new ECC node in T with weight sc(u, v), and add r_u and r_v as its children;
10     else
11         Without loss of generality, assume r_u is an ECC node with weight sc(u, v), and add r_v as a child of r_u in T;
```

4.2 Constructing the hierarchy tree

Given the Steiner connectivities of all edges of a graph G, Algorithm 4 constructs the hierarchy tree of ECO-decomposition of G in a bottom-up manner. The main idea is as follows. First, the hierarchy tree $\mathcal{T}$ is initialized as a forest of singleton nodes. Then, for each edge $(u, v) \in E(G)$ in non-increasing order regarding $sc(\cdot, \cdot)$, we identify the tree in $\mathcal{T}$ (specifically, the root r_u of the tree) containing u and the tree (specifically, the root r_v of the tree) containing v. If u and v are already in the same tree (i.e., $r_u = r_v$), then we do nothing. Otherwise, we merge the two trees into one in $\mathcal{T}$, with the root of this newly formed tree having weight $sc(u, v)$.

The pseudocode of constructing the hierarchy tree is illustrated in Algorithm 4, denoted by ConstructHierarchy. The input of the algorithm is a graph G with $sc(u, v)$ precomputed

for each edge (u, v). It first initializes an empty hierarchy tree (Line 1) and creates a single-node tree in $\mathcal{T}$ for each vertex of G (Line 2). Then, the trees in $\mathcal{T}$ will be merged with each other to form ECC nodes in the hierarchy tree. For each edge $(u, v) \in E(G)$ sorted by $sc(u, v)$ in non-increasing order (Line 3), the roots of the trees in $\mathcal{T}$ containing node u and node v are found, represented by r_u and r_v, respectively (Line 4). If $r_u = r_v$, it implies that vertices u and v have already been merged into the same tree so that the algorithm skips the current edge (Line 5); otherwise, the algorithm merges r_u and r_v into a single tree based on the following three cases. (1) If both r_u and r_v are ECC nodes with weight $sc(u, v)$, it merges r_u and r_v into a single ECC node (Lines 6–7). (2) If neither r_u nor r_v is an ECC node with weight $sc(u, v)$, it creates a new ECC node in $\mathcal{T}$ with weight $sc(u, v)$ whose children are r_u and r_v (Lines 8–9). (3) The last situation is that one of r_u or r_v is an ECC node with weight $sc(u, v)$ and the other is not; note that, if the other one is an ECC node, then its weight must be larger than $sc(u, v)$. Assume that r_u is the one with weight $sc(u, v)$, r_v would be added as a child of r_u. Similar steps would be applied to the situation where r_v is the one with weight $sc(u, v)$ (Lines 10–11).

The most time-consuming operation in Algorithm 4 is Line 4, which aims to find the root of the tree that contains a node u in a forest $\mathcal{T}$. A naive implementation of this operation would take $\mathcal{O}(n)$ time by tracing the parent pointers starting from node u in the tree, and then the total time complexity of Algorithm 4 would be $\mathcal{O}(n \times m)$. This can be improved to $\mathcal{O}(m)$ by resorting to the disjoint-set data structure. Recall that a disjoint-set data structure $\mathcal{D}$ partitions a universe of elements into a collection of sets, and each set is represented by one of its element (called *representative*) [17]. There are two operations supported by the data structure $\mathcal{D}$: Find the set that contains a specific element and merge two sets into one. In our case, the universe of the data structure $\mathcal{D}$ is the set of leaf nodes of the hierarchy tree $\mathcal{T}$, and there is a one-to-one correspondence between sets in $\mathcal{D}$ and trees in $\mathcal{T}$. Whenever we merge two trees in $\mathcal{T}$, we also union the two corresponding sets in $\mathcal{D}$. Moreover, we point each set (specifically, the representative element of the set) of $\mathcal{D}$ to the root of the tree in $\mathcal{T}$ to which the set corresponds. This pointer is used for efficiently identifying the root of the tree that contains a node (i.e., Line 4). As each of the two operations on $\mathcal{D}$ takes amortized constant time [17] [4] and sorting the edges at Line 3 can be achieved in linear time by counting sort [17], the time complexity of Algorithm 4 is $\mathcal{O}(m)$.

[4] To be more precise, the amortized time complexity of each operation on $\mathcal{D}$ is the inverse of the Ackermann function of n [17]. As this function grows very slowly and is bounded by 4 for all practical values of n, we consider it as a constant.

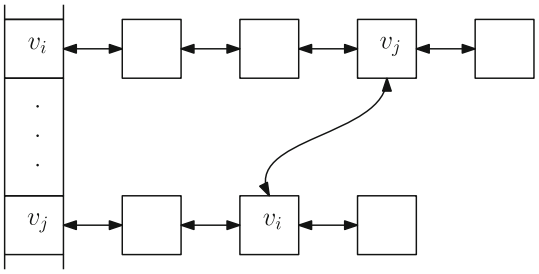


Fig. 9 Doubly linked list-based graph representation

5 Optimizing the space usage

A straightforward implementation of Algorithm 3 would result in a space complexity of $\mathcal{O}((m + n) \log \delta(G))$, i.e., each level of the recursion tree of Fig. 7 would require storing a separate copy of the input graph G. This space complexity is too high for large graphs. In this section, we focus on optimizing the space usage of DC. We first in Sect. 5.1 discuss how to implement DC in $\mathcal{O}(m + n \log \delta(G))$ space by using doubly linked list-based graph representation, where the constant hidden by the big-$\mathcal{O}$ notation is large. Then, in Sect. 5.2 we further optimize the space usage of DC by using adjacency array-based graph representation and other non-trivial optimizations; this results in our space-efficient algorithm DC-AA that has space complexity $2m + \mathcal{O}(n \log \delta(G))$. As a result, we can process billion-scale graphs with an ordinary PC. For example, experiments in Sect. 7 show that our adjacency array-based algorithms can process twitter-2010 and com-friendster, which have 1.2 and 1.8 billion undirected edges, respectively, with at most 15 GB and 24 GB main memory. In contrast, the linked list-based algorithms run out-of-memory even with 128 GB memory.

5.1 Doubly linked list-based implementation

In this subsection, we discuss how to implement DC by using doubly linked list-based graph representation, which is also the representation used by the state-of-the-art KECC algorithm [11] and the two state-of-the-art ECO-decomposition algorithms [7, 42]. The main reason for the existing approaches to choose this representation is that KECC iteratively modifies the graph—i.e., contract two (super-)vertices into one and remove (super-)vertices of degree less than k [11]—which can be easily implemented by using the linked list-based graph representation. We abstract these two graph modification operations as *vertex contraction* and *vertex removal*, respectively. Note that TD also uses the vertex removal operation (see Lines 11–13 of Algorithm 1) and BU uses the vertex contraction operation (see Line 8 of Algorithm 2).

Recall that the linked list-based graph representation stores the adjacent edges of each vertex in a linked list [17]. For example, Fig. 9 illustrates the linked lists for the adjacent edges of v_i and v_j. In addition, a cross pointer is constructed in the implementation for each edge (v_i, v_j) which points to its reverse direction (v_j, v_i), as each undirected edge will have two copies in the representation, one copy for each direction. Vertex removal can be implemented efficiently as follows. Suppose we are removing vertex v_i from the graph; note that we also need to remove all edges ending at v_i which scatter across the linked lists. To achieve this, we iterate through all the adjacent edges of v_i, and for each edge (v_i, v_j), we first locate its reverse edge (v_j, v_i) *via the cross pointer* and then remove (v_j, v_i) from the doubly linked list of v_j which can be achieved in constant time. When it comes to vertex contraction, the process becomes slightly more complicated. Suppose we are contracting v_i and v_j. We use one of the vertices (e.g., v_i) to represent the resulting super-vertex, and the process is divided into two parts: the edges starting from v_j should start from v_i; the edges end at v_j ought to end at v_i. For the first part, we could simply connect the head of the linked list of v_j to the tail of the linked list of v_i. For the second part, we iterate through all the adjacent edges of v_j, and for each edge (v_j, v_k), we first locate its reverse edge (v_k, v_j) via the cross pointer and then update the edge to be (v_k, v_i).

Based on the linked list-based graph representation, DC (i.e., Algorithm 3) can be implemented fairly easily. Specifically, to construct $g_1 = \mathsf{GS}_L^{M-1}(g)$ and $g_2 = \mathsf{GS}_M^H(g) = \phi_M(g)$ from $g = \mathsf{GS}_L^H(G)$ at Line 10 of Compute-DC, we first split each linked list (that corresponds to the adjacent edges of a vertex) into two, one to be used in g_1 and the other in g_2, as g_1 and g_2 have disjoint sets of edges. We then apply the contraction operation for the edges in g_1. In this way, we do not create any new edges in Compute-DC; note, however, that the number of vertices may double (i.e., one copy in g_1 and one in g_2). Overall, DC has a space complexity of $\mathcal{O}(m + n \log \delta(G))$, by noting that it traverses the recursion tree of Fig. 7 in a depth-first manner.

5.2 Adjacency array-based implementations

Although the space complexity of DC has been reduced from $\mathcal{O}((m+n)\log\delta(G))$ to $\mathcal{O}(m + n\log\delta(G))$ in Sect. 5.1, this is still too high to be applied to large graphs (see our experimental results in Sect. 7) as the constant hidden by the big-$\mathcal{O}$ notation is large. Firstly, for each edge in the linked lists, three pointers and one number (where the number indicates the other end-point of the edge) need to be stored. Thus, the graph representation will consume at least $8m$ integers, by noting that each undirected edge is stored twice. Secondly, the graph may be stored three times (i.e., simultaneously have three copies in main memory) during the computation, i.e., once in Compute-DC and twice in KECC as KECC will modify the graph that is input to it [11]. In this subsection, we propose an adjacency array-based implementation to explicitly bound the constant on m by 2 such that the space complexity becomes $2m + \mathcal{O}(n \log \delta(G))$, and at the same time keep the time complexity unchanged which is challenging. Note that we do not optimize the constant on $n \log \delta(G)$, as real-world graphs usually have much more edges than vertices, i.e., m usually is the dominating factor.

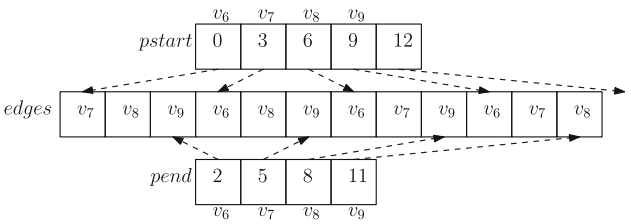


Fig. 10 Adjacency array-based graph representation

The adjacency array-based graph representation is also known as the compressed sparse row (CSR) representation. It uses two arrays to represent a graph, and assumes that the vertices are taking ids from $\{0, \ldots, n-1\}$. We denote the two arrays by $pstart$ and $edges$. The set of adjacent edges (specifically, neighbors) of each vertex is stored consecutively in an array, and then, all such arrays are concatenated into the large array $edges$. The start position of the set of adjacent edges of vertex i in $edges$ is stored in $pstart[i]$, and thus, the set of adjacent edges of vertex i is stored consecutively in the subarray $edges[pstart[i], \ldots, pstart[i+1]-1]$. Figure 10 demonstrates such a representation for the subgraph g_2 of Fig. 2; please ignore the part of "$pend$" for the current being. The array $pstart$ is of size $n+1$, while the array $edges$ is of size $2m$.

Efficient Implementation of Vertex Removal and Contraction. To achieve the space complexity of $2m + \mathcal{O}(n \log \delta(G))$, we will not be allowed to create any new copies of $edges$, even if temporarily. This makes it challenging to efficiently implement vertex removal and vertex contraction which are the two primitive operations used by the algorithms. In the following, we discuss how to implement these two operations efficiently with the help of some additional data structures of size $\mathcal{O}(n)$.

Vertex removal in the adjacency array-based graph representation can be implemented by marking the vertex as "removed". Recall that, when vertex i is "removed", the edge (j, i) that ends at i should also be removed from the adjacent edges of j for each neighbor j of i. This cannot be implemented efficiently without cross pointers, but *storing cross pointers is not affordable for achieving the space complexity of* $2m + \mathcal{O}(n \log \delta(G))$. To circumvent this, we propose to remove (j, i) from the adjacent edges of j in a *lazy* way, i.e., delay it to the moment when we actually need to traverse all

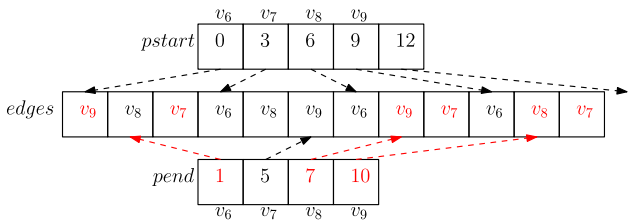


Fig. 11 After removing vertex v_7

adjacent edges of j. Thus, we introduce another array, named $pend$, of size n, where the entry $pend[j]$ explicitly stores the last position of the adjacent edges of vertex j in $edges$ and is initialized with $pstart[j+1]-1$; see Fig. 10. When we need to traverse all the adjacent edges of j, we loop through all the index values idx from $pstart[j]$ to $pend[j]$: If the edge $edges[idx]$ should have been removed (i.e., the other end-point of this edge is "removed"), we first swap $edges[idx]$ with $edges[pend[j]]$ and then decrement $pend[j]$ by one. In this way, all the remaining (i.e., active) adjacent edges of vertex j would be consecutive in $edges$ starting from position $pstart[j]$ and ending at $pend[j]$, while the edges in $edges$ whose indices are between $pend[j]+1$ and $pstart[j+1]-1$ are "removed". Thus, the amortized time of removing an edge is constant. For example, the result of removing vertex v_7 from the graph of Fig. 10 is shown in Fig. 11; here, for illustration purpose, we assume that the graph has been traversed once such that $edges$ is reorganized.

When contracting vertex i and vertex j, following the same ideas as Sect. 5.1 we also use v_i to represent the resulting super-vertex and divide the process into two parts: the edges starting from v_j should start from v_i; the edges ending at v_j ought to end at v_i. For the first part, instead of moving adjacent edges around which would create temporary copies of $edges$ and furthermore increase the time complexity, we use two additional arrays, sv_next and sv_last, each of size n to represent the super-vertices. That is, sv_next chains together all vertices that belong to the same super-vertex, implicitly represented as a singly-linked list; specifically, $sv\_next[i]$ stores the id of the next vertex (i.e., after i) in the super-vertex. To efficiently merge two super-vertices (that are represented as singly-linked lists), we also store in $sv\_last[i]$ the id of the last vertex in the super-vertex i. For example, Fig. 12a shows the values of sv_next and sv_last for the graph of Fig. 10; note that the part in the dotted rectangle illustrates the linked lists that represent the super-vertices and is not physically stored. When contracting (super-)vertex i with (super-)vertex j, we first update $sv\_next[sv\_last[i]]$ to j to connect the two linked lists into one, and then update $sv\_last[i]$ to $sv\_last[j]$; this can be conducted in constant time. Note that $sv\_last[i]$ is only useful and up-to-date if i is the first vertex in a linked list, i.e., $sv\_last[\cdot]$ for all other vertices are not updated and will

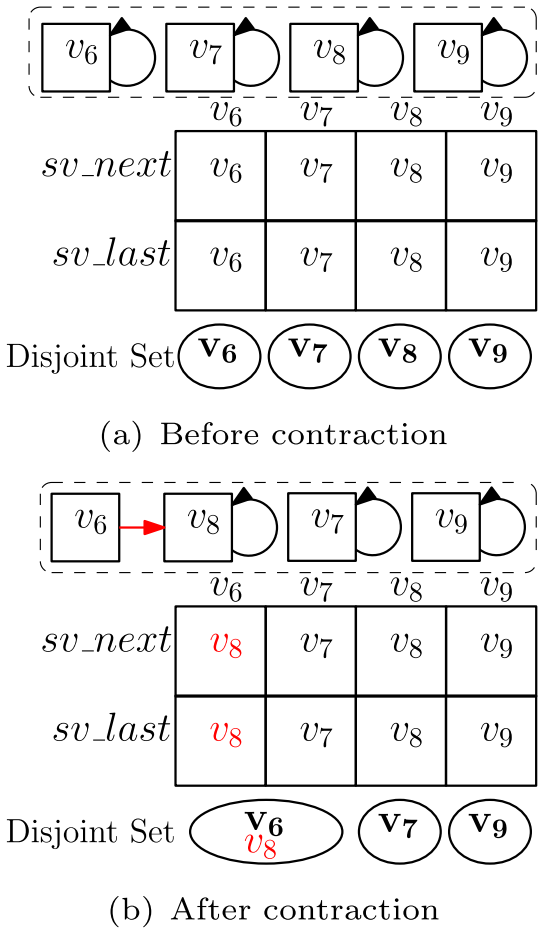


Fig. 12 Example of contracting v_6 and v_8

not be used. Figure 12b shows the result of contracting v_6 and v_8; notice that v_6 and v_8 are now linked together. To iterate over all edges adjacent to (super-)vertex i, we use a pointer p which is initialized as i and is then iteratively updated by $sv\_next[p]$ until reaching the end of the linked list. These p values correspond to ids of the vertices that are contracted into (super-)vertex i. Thus, the edges adjacent to (super-)vertex i are $edges[pstart[p], \ldots, pend[p]]$ for all p values along the iterations.

For the second part of vertex contraction (i.e., edges ending at v_j ought to end at v_i), explicitly modifying the edge end-points without maintaining cross pointers would be time-consuming. To tackle this issue, we propose to use an additional disjoint-set data structure of size $\mathcal{O}(n)$ to represent the super-vertices. The universe of the data structure is the vertex set V, and each super-vertex corresponds to a set in the data structure that consists of the vertices contained in the super-vertex. When we contract two super-vertices, we also union their corresponding sets in the data structure. In addition, we point the representative of a set in the data structure to the vertex that represents the corresponding super-vertex, in the same way as that in constructing the hierarchy tree

in Sect. 4.2. The last row of Fig. 12 illustrates the disjoint sets, where the representative of a set is shown in bold, e.g., v_6 and v_8 are in the same set in Fig. 12b with v_6 being the representative.

Our Space-Optimized Algorithms. With the ideas presented above, we first optimize the space usage of KECC by using the adjacency array-based graph representation, as it is an essential procedure used in DC. We denote our space-optimized version of KECC as KECC-AA. Note that, with the above implementations of vertex removal and vertex contraction, the input graph to KECC-AA is always represented by $pstart$ and $edge$ which are not changed, although the order of the adjacent edges for each vertex may change. Thus, we do not need to store another copy of the input graph, and the space complexity of KECC-AA is $2m + \mathcal{O}(n)$.

With KECC-AA, we are now ready to present our space-optimized version of DC. It is worth pointing out that directly replacing KECC with KECC-AA in Algorithm 3 will not achieve our desired space complexity. The main idea is still based on the fact that g_1 and g_2 in Algorithm 3 have disjoint sets of edges. But now, we cannot afford to first construct g_1 and g_2 from g and then release the memory of g, as this will double the intermediate memory consumption. To tackle this issue, we always expand the right child of a node in the recursion tree (see Fig. 7) before expanding the left child. This is based on the observation that, for a non-leaf node in the recursion tree, the graph processed by its right child is always a subgraph of the current graph, while the graph processed by the left child is obtained by contracting each connected component (of the graph of the right child) into a super-vertex in the current graph. Thus, to process the right child, we can directly work on $pstart$ and $edges$ by rearranging the adjacent edges of each vertex and using a *local* array of size n to bookmark the number of adjacent edges of each vertex in the subgraph. After expanding the right child (and its descendants) and to process the left child, we further create a *local* copy of sv_next, sv_last and the disjoint-set data structure, which are all of size $\mathcal{O}(n)$, to implement the contraction operation.

The pseudocode of the adjacency array-based implementation of DC is illustrated in Algorithm 5, denoted by DC-AA. It is generally similar to Algorithm 3, with three differences. Firstly, it invokes KECC-AA instead of KECC at Line 10. Secondly, it expands the right child first (Lines 11–13). Thirdly, it interleaves the execution of Algorithm 4 with Construct-DC-AA (Lines 2, 3, 7). The reason of interleaving is that explicitly storing the Steiner connectivities of all edges would increase the space consumption by at least $2m+\mathcal{O}(n)$, and interleaving eliminates the requirement of storing the Steiner connectivities. This interleaving is correct because the right child is always expanded before the left child for each node in the recursion tree (Fig. 7), and thus, the Steiner connectivities are computed in non-increasing order. Note that *we also exploit this interleaving to reduce the memory consumption for* DC, TD*and* BU*in our experiments*.

Algorithm 5: DC-AA(G)

```
1  Compute the degeneracy δ(G) of G;
2  Execute Lines 1–2 of Algorithm 4;
3  Construct-DC-AA(G, 1, δ(G));
4  return T;

   Procedure Construct-DC-AA(g, L, H)
5  if L = H then
6      foreach edge (u, v) ∈ E(g) do
7          Execute Lines 4–11 of Algorithm 4 with sc(u, v) equal to L;
8  else
9      M ← ⌈(L+H)/2⌉;
10     φ_M(g) ← KECC-AA(g, M);
11     foreach connected subgraph g' ∈ φ_M(g) do
12         Construct-DC-AA(g', M, H);
13         Contract g' into a super-vertex in g;
14     Construct-DC-AA(g, L, M − 1);
```

The correctness of DC-AA directly follows from the correctness of the discussions in the above two paragraphs, and the time complexity of DC-AA remains the same as DC since our adjacency array-based implementation does not increase the time complexity of vertex removal and contraction. The space complexity of DC-AA becomes $2m+\mathcal{O}(n \log \delta(G))$, as it conducts a depth-first traversal of the recursion tree (Fig. 7) and each level of the recursion tree only requires a local data structure of size $\mathcal{O}(n)$.

With the same idea as DC-AA, we can also implement TD and BU by using the adjacency array-based graph representation such that their space complexities become $2m+\mathcal{O}(n)$, while their time complexities remain unchanged. We denote our space-optimized versions of TD and BU by TD-AA and BU-AA, respectively.

6 A practically efficient approach

In this section, we propose a simpler and practically efficient approach for ECo-decomposition by optimizing BU-AA. Our empirical studies in Sect. 7.1.1 (specifically, Fig. 15) show that BU-AA can outperform DC-AA when the degeneracy $\delta(G)$ is small (e.g., less than 25 or so), but is outperformed by DC-AA when $\delta(G)$ becomes large. The main reason of BU-AA outperforming DC-AA for small $\delta(G)$ is that (1) the difference between $\delta(G)$ and $\log \delta(G)$ is small when $\delta(G)$ is small and (2) BU-AA is simpler and thus has a smaller constant coefficient in the time complexity analysis. When $\delta(G)$ becomes larger, BU-AA is outperformed by DC-AA because BU-AA needs to invoke KECC on G (or more precisely, a

contracted version of G) for a large number (i.e., $\delta(G)$) of iterations (see Line 4 of Algorithm 2). We propose a simple optimization technique, called k-core reduction, to improve the efficiency of Line 4 of Algorithm 2 by significantly reducing the graph that is input to KECC when $\delta(G)$ is large. The definition of k-core is as follows.

Definition 3 (k-core [34]) Given a graph G and an integer k, the k-core of G is the *maximal* subgraph g of G such that the minimum degree of g is at least k.

Like k-ECC, k-core is also a vertex-induced subgraph. For example, for the graph in Fig. 2, the entire graph is a 3-core, and g_1 as induced by vertices $\{v_1, v_2, v_3, v_4, v_5\}$ is a 4-core.

A related concept is *core number*, denoted $\mathsf{core}(\cdot)$, where $\mathsf{core}(u)$ is the largest k such that u is included in the k-core. For example, for the graph in Fig. 2, the core numbers for $\{v_1, v_2, v_3, v_4, v_5\}$ are 4, while the core numbers for all other vertices are 3. It is known that the k-core is simply $G[\{v \mid v \in V(G) \text{ s.t. } \mathsf{core}(v) \geq k\}]$, i.e., the subgraph of G induced by the set of all vertices with core number at least k [9].

The following lemma claims the relationship between the k-core and k-ECCs of a graph.

Lemma 3 [11] *Given a graph G and an integer k, all k-ECCs of G are subgraphs of the k-core of G.*

For example, for the graph in Fig. 2, $g_1 \cup g_2$ and g_3 are the two 3-ECCs of the graph and g_1 is the only 4-ECC.

Following Lemma 3, when computing k-ECCs, it is sufficient to restrict the computation to the k-core of the graph, which could be much smaller than the entire graph. We call this process the *k-core reduction*. It is worth pointing out that although the k-core reduction has already been utilized in the existing algorithms TD and BU, more specifically in KECC-AA, the full potential of k-core reduction has not been exploited. This is because KECC-AA applies the k-core reduction to reduce its input graph, and although the k-core can be computed in time linear to the number of edges [4, 28], the input graph to KECC-AA could be much larger than the k-core. In view of this, we propose to optimize BU-AA by restricting the input of KECC-AA to be the k-core of the input graph, or more precisely a contracted version of the k-core.[5]

The pseudocode of our optimized BU-AA is shown in Algorithm 6; we denote the optimized version by BU*-AA. The approach is very similar to Algorithm 2, with the only differences at Lines 2–5 and 7–8. Specifically, KECC-AA is now invoked on a subgraph G' of G instead of on the entire graph G at Line 8, where G' is extracted at Line 7. Moreover, the subgraph G' is extracted from G in time that is linear to the number of edges of G', rather than linear to the number of edges of G. This is achieved through Lines 2–4, details of which will be introduced shortly.

Algorithm 6: BU*-AA(G)

```
1  Create one leaf node in T for each vertex of G;
2  core(·) ← CoreDecomposition(G);
3  Reorganize the adjacency edges of each vertex;
4  Sort the vertices in non-increasing order regarding their core
   values;
5  δ(G) ← max_{u∈V} core(u);
6  for k ← δ(G) down to 1 do
7      G' ← the subgraph of G induced by vertices whose core
       numbers are at least k;
8      φ_k(G') ← KECC-AA(G', k);
9      for each connected subgraph g ∈ φ_k(G') do
10         Create an ECC node ecc in T with weight k;
11         Add the set of nodes of T that correspond to vertices of g
           to be the children of ecc;
12         Contract g into a single super-vertex in G, to which ecc
           corresponds;
13 return T;
```

In accordance with Lemma 3, $\phi_k(G')$ computed in Algorithm 6 is exactly the same as $\phi_k(G)$ computed in Algorithm 2 for all possible k values. Thus, the hierarchy tree constructed by invoking BU*-AA is the same as the one constructed by invoking BU-AA. Based on the definition of k-core, the k_1-core of a graph must be a subgraph of the k_2-core of the graph if $k_1 \geq k_2$. Therefore, when $k = \delta(G)$, G' is the $\delta(G)$-core of G, which is typically only a small part of the entire graph. After decrementing k by 1, G' is then expanded to the $(\delta(G) - 1)$-core of G. So on so forth, G' would finally be expanded to the whole graph when k gets to some smaller number. Nevertheless, along the expansion of G', k-ECCs are contracted into super-vertices, which reduces the size of G and thus the extracted G'. Intuitively, KECC-AA would seldom be invoked on an extremely large graph based on the analysis. Hence, BU*-AA is expected to perform much more efficiently compared to BU-AA in practice. However, note that the worst-case time complexity of BU*-AA remains the same as that of BU-AA.

Example 4 Here, we apply BU*-AA on the graph G in Fig. 2 as an example. Core values for $\{v_1, v_2, v_3, v_4, v_5\}$ are computed as 4, and core values for the other vertices in G are computed as 3; thus, $\delta(G)$ is 4. Initially, 13 leaf nodes are created in the hierarchy tree $\mathcal{T}$ for the thirteen vertices of G. During the first iteration, G' is the 4-core of G, which is equivalent to the subgraph g_1. It could be found that $\phi_4(g_1) = g_1$ so that an ECC node of weight 4 with five children v_1, v_2, v_3, v_4, v_5 is added into $\mathcal{T}$. g_1 is contracted into a single super-vertex denoted by s_1. Then, in the second iteration, G' is the entire

[5] Note that it is in principle also possible to optimize TD-AA and DC-AA by the k-core reduction. However, this is much more complicated and may be not efficient, e.g., by noting that the input to KECC at Line 4 of Algorithm 1 could be even smaller than the k-core of G. Thus, we only optimize BU-AA by the k-core reduction in this paper.

graph where g_1 is contracted into a single super-vertex s_1; it could be obtained that $\phi_3(G') = \{s_1 \oplus g_2, g_3\}$. As a result, two ECC nodes of weight 3 are added into $\mathcal{T}$. Similarly, $s_1 \oplus g_2$ are contracted into a super-vertex denoted by $s_{1,2}$, and g_3 is contracted into a super-vertex denoted by s_3. In the third iteration, we have $\phi_2(G') = \{s_{1,2} \oplus s_3\}$ given the 2-core of G is G itself. Finally, we obtain the same hierarchy tree as shown in Fig. 3.

Efficiently Extracting k-core. Recall that Line 8 of Algorithm 6 processes the k-core of a contracted version of G; note that subgraphs are contracted into super-vertices at Line 12. To facilitate an efficient extraction of the k-core, we propose to compute the core number $\mathsf{core}(\cdot)$ for each vertex of G (Line 2) and reorganize the edges and vertices in the graph representation based on the computed core numbers (Lines 3–4). As a result, the k-core can be extracted in time linear to the size of the k-core (Line 7).

Recall that the adjacent edges (or neighbors) of a vertex v are stored consecutively in the subarray $edges[pstart[v], \ldots, pstart[v+1]-1]$. Specifically, this subarray is reorganized in two steps as follows:

1. **Split the subarray into two parts.** We first split the subarray into two parts by putting all its neighbors whose core numbers are at least $\mathsf{core}(v)$ at the front (referred to as **Part1**), followed by the other neighbors (referred to as **Part2**). Note that, in this step, the order within each part is arbitrary. This is achieved by using two pointers p_1 and p_2 which are initialized as $pstart[v]$ and $pstart[v+1]-1$, respectively. Then, as long as $\mathsf{core}(edges[p_1])$ is no less than $\mathsf{core}(v)$, we increment p_1 by 1, and as long as $\mathsf{core}(edges[p_2])$ is smaller than $\mathsf{core}(v)$, we decrement p_2 by 1. Otherwise, $\mathsf{core}(edges[p_1]) < \mathsf{core}(v)$ and $\mathsf{core}(edges[p_2]) \geq \mathsf{core}(v)$; we swap $edges[p_1]$ with $edges[p_2]$. This process terminates when the two pointers meet each other. In this case, $edges[pstart[v], \ldots, p_1 - 1]$ is **Part1** and $edges[p_1, \ldots, pstart[v+1]-1]$ is **Part2**.
2. **Sort the neighbors in Part2 into non-increasing order regarding their core numbers.** We achieve this by using bin sort.

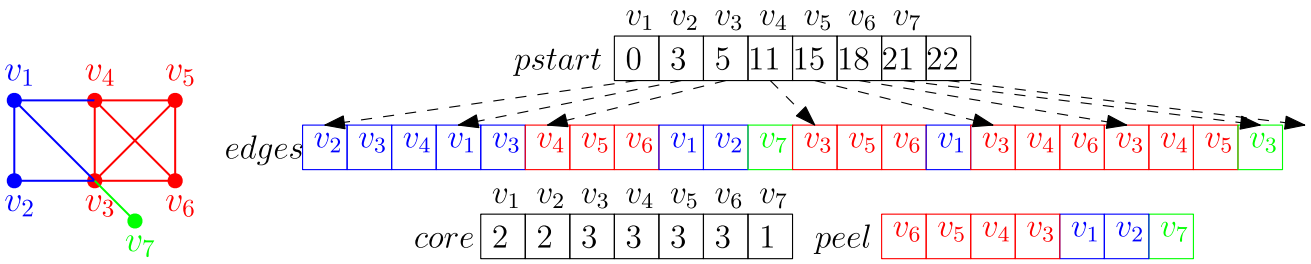
Let $deg(v)$ be the degree of v in G. It is easy to see that the time complexity of Step 1 is $\mathcal{O}(deg(v))$ and the time complexity of Step 2 is $\mathcal{O}(deg(v) + \mathsf{core}(v)) = \mathcal{O}(deg(v))$; note that for the bucket sort in Step 2, the maximum key value is $\mathsf{core}(v)$. Thus, the overall time complexity of organizing the adjacent edges of all vertices is $\mathcal{O}(\sum_{v \in V} deg(v)) = \mathcal{O}(m)$.

With vertices and adjacent edges in order, we can extract the k-core in time linear to the size of the k-core regardless of the size of the graph, as follows. The k-core contains all vertices with core number larger than or equal to k, which can be extracted by scanning vertices in non-increasing core number order until a vertex with core number lower than k is encountered. Then, edges of the k-core can be recovered by looking into the neighbors of each extracted vertex. That is, for each extracted vertex u, we scan its neighbors from the beginning until meeting a neighbor with core number lower than k. As all neighbors in **Part1** of the neighbors of u are guaranteed to be included in the k-core whenever u is in the k-core, we did not enforce any order for the **Part1** neighbors of each vertex when reorganizing the edges.

Example 5 Figure 13 comes up with a detailed example indicating the intuition of extracting the k-core for different k values with our adjacency array-based graph representation. Two arrays, *pstart* and *edges* in Fig. 13b, represent the example graph in Fig. 13a. The array *core* stores the core number of all vertices as computed by core decomposition on the example graph. The *peel* array sorts vertices in non-increasing order regarding their core numbers; note that this *peel* array actually can be directly obtain from the core decomposition, which peels vertices in the reverse order.

Neighbors in *edges* are ordered in the correct order according to their core numbers. It is worth pointing out that v_2 can be ahead of v_3 in the *edges* array within v_1's neighbors despite that the core number of v_3 is larger than the core number of v_2, since the core numbers of both vertices are larger than or equal to the core number of v_1. On the other hand, v_1, v_2, and v_7 have to be arranged in descending order according to their core numbers in v_3's neighbors because their core numbers are smaller than the core number of v_3. In this case, the 3-core of the graph is composed by vertices and edges marked red. Then, the 2-core of the graph is made up of vertices and edges marked blue, together with the 3-core. Finally, the 1-core is the graph itself, which is green part in addition to the 2-core.

Complexity Analysis. Here, we analyze the time complexity of Lines 2–4 and 7 of Algorithm 6, which are the lines added in addition to Algorithm 2. Firstly, Line 2 takes $\mathcal{O}(m)$ time by the peeling algorithm [4, 28], which iteratively removes the vertex with the minimum degree from the graph. Secondly, Line 3 can be conducted in $\mathcal{O}(m)$ time as discussed above. Thirdly, Line 4 can be conducted in $\mathcal{O}(n)$ time by bin sort. Lastly, Line 7 can be conducted in time that is linear to the size of the extracted graph G' by noting that for each vertex u in G' we access at most one of its adjacent edges that is not in G'. Thus, the time complexity of Algorithm 6 is the same as that of Algorithm 2, i.e., $\mathcal{O}(\delta(G) \times \mathsf{T}_{\mathsf{KECC}}(G))$. Compared with Algorithm 2, the additional computations of Algorithm 6 can be conducted very efficiently, and the input to KECC-AA in Algorithm 6 can be much smaller than that in Algorithm 2; hence, the optimization of Algorithm 6 can bring an enormous improvement in practice.



(a) Example graph (b) k-core extraction of the example graph with adjacency array-based graph representation

Fig. 13 k-core extraction on an example graph

Regarding the space complexity, we note that Algorithm 6 can be implemented to take space $2m+\mathcal{O}(n)$. To achieve this, we do not actually explicitly extract and store the subgraph G' at Line 7. Instead, whenever we need to access the neighbors of a vertex u in G', we access the longest prefix of the adjacent edges in $edges[pstart[u], \ldots, pstart[u+1]-1]$ such that the core numbers of the end-points are at least k.

7 Experiments

In this section, we conduct extensive performance studies to evaluate the efficiency and effectiveness of our techniques. Specifically, we evaluate the following ECo-decomposition algorithms:

- TD (Algorithm 1): the existing top-down approach proposed in [7] that uses the doubly linked list-based graph representation.
- BU (Algorithm 2): an adaptation of the existing bottom-up approach proposed in [42] that uses the doubly linked list-based graph representation.
- DC (Algorithm 3): our near-optimal approach (Algorithm 3) that uses the doubly linked list-based graph representation and has a space complexity of $\mathcal{O}(m + n \log \delta(G))$.
- TD-AA, BU-AA, and DC-AA: space-optimized versions of TD, BU and DC by using the adjacency array-based graph representation (Sect. 5.2).
- BU*-AA (Algorithm 6): our core optimized version of BU-AA.

In addition, we also evaluate two k-ECC computation algorithms:

- KECC: the state-of-the-art algorithm proposed in [11] that uses the doubly linked list-based graph representation.
- KECC-AA: our space-optimized version of KECC that uses the adjacency array-based graph representation (Sect. 5.2).

The source code of our algorithms is available at https://lijunchang.github.io/ECo-Decompose/. All algorithms are implemented in C++ and compiled with GNU GCC with the -O3 optimization. All experiments are conducted on a machine with Intel(R) Xeon(R) 3.6GHz CPU and 128 GB memory running Ubuntu. We evaluate the performance of all algorithms on both real and synthetic graphs as follows.

Real Graphs. We evaluate the algorithms on fourteen real graphs from different domains, which are downloaded from the Stanford Network Analysis Platform [6] and the Laboratory of Web Algorithmics .[7] Statistics of the graphs are shown in Table 1, where the second last column and the last column, respectively, show the average degree and the degeneracy. The graphs are ranked regarding their numbers of edges. We denote the graphs by D1, . . ., D14.

Synthetic Graphs. We also evaluate the algorithms on two kinds of synthetic graphs that are generated by the graph generator GTGraph .[8]

- *Power-law graphs*: A power-law graph is a random graph in which edges are randomly added such that the degree distribution follows a power-law distribution.
- *SSCA graphs*: An SSCA graph contains a collection of randomly sized cliques and also random inter-clique edges.

Firstly, we generate 14 power-law graphs, PL1, . . ., PL14, where the number of vertices varies from 16 thousand to 133 million with an increasing factor of 2. The average degree of the power-law graphs is around 24.5; as a result, the number of undirected edges of the power-law graphs varies from 198 thousand to 1.6 billion. The degeneracy of these graphs varies from 18 to 25.

Secondly, we further generate six power-law graphs fixing the number of vertices to be the same as PL7 (i.e., around one million), PL7_1, . . ., PL7_6, where the number of edges

[6] http://snap.stanford.edu/.

[7] http://law.di.unimi.it/datasets.php.

[8] http://www.cse.psu.edu/~madduri/software/GTgraph/.

Table 1 Statistics of graphs

ID	Dataset	m	n	$\bar{d}$	δ
D1	ca-CondMat	91,286	21,363	8.55	25
D2	soc-Epinions1	405,739	75,877	10.69	67
D3	web-Google	3,074,322	665,957	9.23	44
D4	as-Skitter	11,094,209	1,694,616	13.09	111
D5	cit-Patents	16,518,947	3,774,768	8.75	64
D6	soc-pokec	22,301,964	1,632,803	27.32	47
D7	wiki-topcats	25,444,207	1,791,489	28.41	99
D8	com-lj	34,681,189	3,997,962	17.35	360
D9	soc-LiveJournal1	42,845,684	4,843,953	17.69	372
D10	com-orkut	117,185,083	3,072,441	76.28	253
D11	uk-2002	261,556,721	18,459,128	28.34	943
D12	webbase	854,809,761	115,554,441	14.79	1,506
D13	twitter-2010	1,202,513,344	41,652,230	57.74	2,488
D14	com-friendster	1,806,067,135	65,608,366	55.06	304

$\bar{d}$, average degree; δ, degeneracy

increases with a factor of 2. The resulting degeneracy of these graphs increases from 21 (for PL7) to 1, 380 (for PL7_6), also roughly with a factor of 2.

Thirdly, we generate 12 SSCA graphs, SSCA1, ..., SSCA12, where the number of vertices varies from 4 thousand to 8 million with an increasing factor of 2. The average degree of the SSCA graphs varies from 12 to 135, and the number of undirected edges varies from 24 thousand to 567 million. The degeneracy of the SSCA graphs varies from 15 to 202.

Evaluation Metrics. For all the evaluations, we record both the processing time and the peak main-memory usage. Each testing is run three times, and the average results are reported. All algorithms are run in main memory and use a single thread. For the reported processing time, we exclude the I/O time that is used for loading a graph from disk to main memory. The peak memory usage of a program is recorded by /usr/bin/time .[9]

7.1 Results for ECo-decomposition

We first compare DC-AA with the existing algorithms in Sect. 7.1.1 and then compare BU*-AA with DC-AA in Sect. 7.1.2.

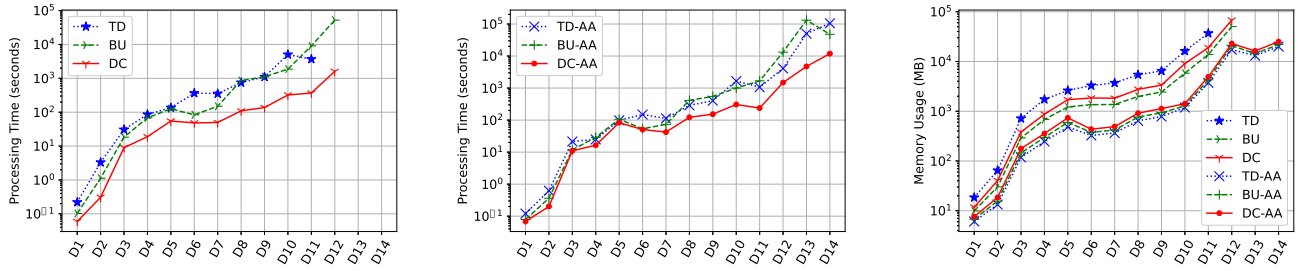
7.1.1 Comparing DC-AA with existing algorithms

In this subsection, we evaluate the six ECo-decomposition algorithms, TD, BU, DC, TD-AA, BU-AA, and DC-AA, regarding their processing time and main-memory usage.

Results on Real Graphs. We first evaluate the algorithms on real graphs. The results are illustrated in Fig. 14. For better comparison, we separate the algorithms into two groups: linked list-based algorithms (i.e., TD, BU, and DC), and space-optimized algorithms (i.e., TD-AA, BU-AA, and DC-AA). The processing time of the three linked list-based algorithms is illustrated in Fig. 14a. We can see that our near-optimal approach DC consistently runs faster than the two state-of-the-art approaches TD and BU, which is inline with our theoretical analysis that the former has a lower time complexity than the latter two. However, all the three algorithms fail to process the two billion-scale graphs D13 and D14, due to running out-of-memory. On the other hand, our space-optimized algorithms are able to process these billion-scale graphs as shown in Fig. 14b, due to their reduced main-memory usage. The overall trend is similar to their counterparts in Fig. 14a, i.e., DC-AA consistently performs the best. When comparing the top-down approach TD-AA with the bottom-up approach BU-AA, there is no clear winner despite of having the same time complexity, as their practical performance is sensitive to the graph topology. For example, the processing time of TD-AA, BU-AA, and DC-AA on D13 is, respectively, 13.9hrs, 36.8hrs, and 1.3hrs, while that on D14 is, respectively, 29.4hrs, 13.3hrs, and 3.3hrs.

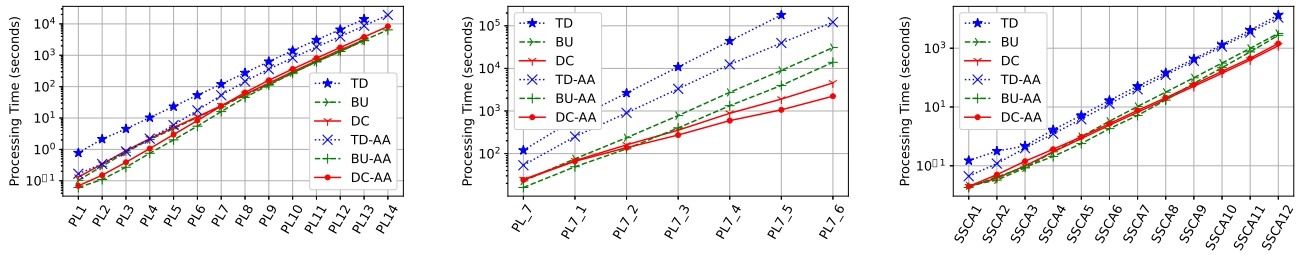
The main-memory usage of the six algorithms is demonstrated in Fig. 14c. It is evident that our space-optimized algorithms (TD-AA, BU-AA, DC-AA) consume much less memory than the linked list-based algorithms (TD, BU, DC), where TD and BU are the two state-of-the-art approaches. For example, the peak memory usage of our space-optimized algorithms is at most 15 GB for D13 and is at most 24 GB for D14, while the linked list-based algorithms run out-of-memory even with 128 GB memory. There are three things

[9] https://man7.org/linux/man-pages/man1/time.1.html.

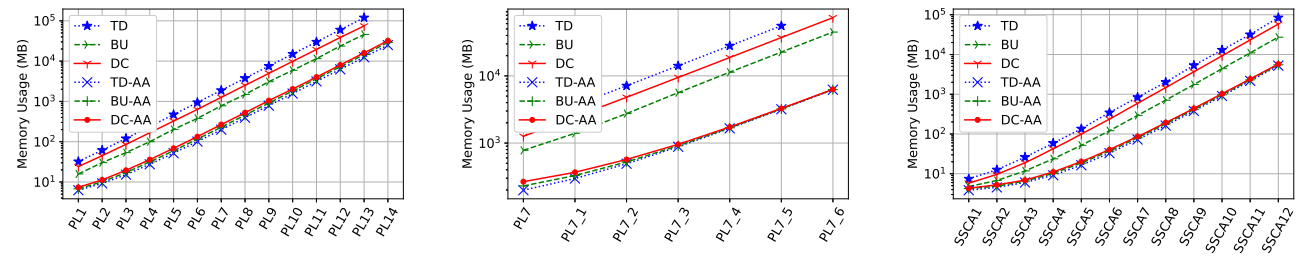


(a) Processing time of linked list-based algorithms (b) Processing time of space-optimized algorithms (c) Memory usage of all algorithms

Fig. 14 Comparing DC-AA with existing algorithms on real graphs (best viewed in color)



(a) Processing time on power-law graphs (vary n and m, fix average degree) (b) Processing time on power-law graphs (vary m, fix n) (c) Processing time SSCA graphs

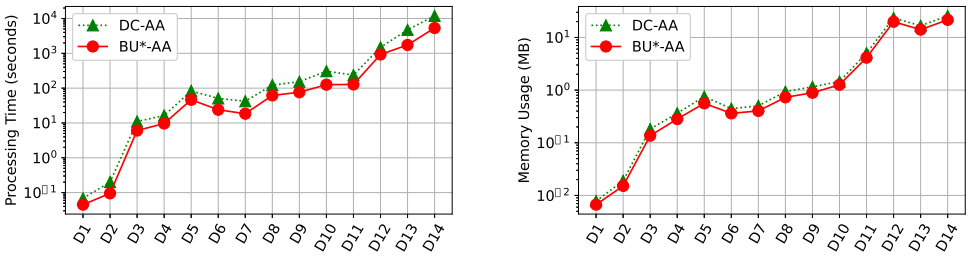


(d) Memory usage on power-law graphs (vary n and m, fix average degree) (e) Memory usage on power-law graphs (vary m, fix n) (f) Memory usage on SSCA graphs

Fig. 15 Comparing DC-AA with existing algorithms on synthetic graphs

worth mentioning for Fig. 14c. Firstly, it appears that TD consumes more memory than DC. This is due to implementation differences, i.e., we used the original implementation of TD from [7] while our implementations of BU and DC slightly optimized the constant on m in the space complexity. We do not optimize the code of TD, as linked list-based implementations, which are outperformed by their space-optimized counterparts, are not our main focus. Secondly, the linked list-based algorithm consume more memory on D13 than on D12, while for our space-optimized algorithms, the situation is the opposite. This is because (1) D13 has more edges but less vertices than D12, (2) the memory usage of linked list-based algorithms is mainly dominated by the part on m in the space complexity, while the memory usage of our space-optimized algorithms is also affected by the part on n. This is also observed for k-ECC computation algorithms in Fig. 19b. Thirdly, DC-AA always consumes a little bit more memory than TD-AA and BU-AA, which is due to the $\log \delta(G)$ factor on n in the space complexity of DC-AA. Nevertheless, the gap is minor given that m dominates $\mathcal{O}(n \log \delta(G))$ for typical graphs.

Results on Synthetic Graphs. The processing time and memory usage of the six algorithms on synthetic graphs are shown in Fig. 15. The overall trend is similar to that on real graphs in Fig. 14. That is, our divide-and-conquer algorithms DC and DC-AA run the fastest, and our space-optimized algorithms consume much less memory than the linked list-based algorithms, e.g., the latter run out-of-memory on PL14 which has 1.6 billion undirected edges. It is interesting to observe that our space-optimized bottom-up approach BU-AA also perform quite well on power-law graphs that have small degeneracy (i.e., at most 25), see Fig. 15a. The results on power-law graphs by varying m and fixing n are shown in Fig. 15b, e; note that the degeneracy of these graphs also



(a) Processing time (b) Memory usage

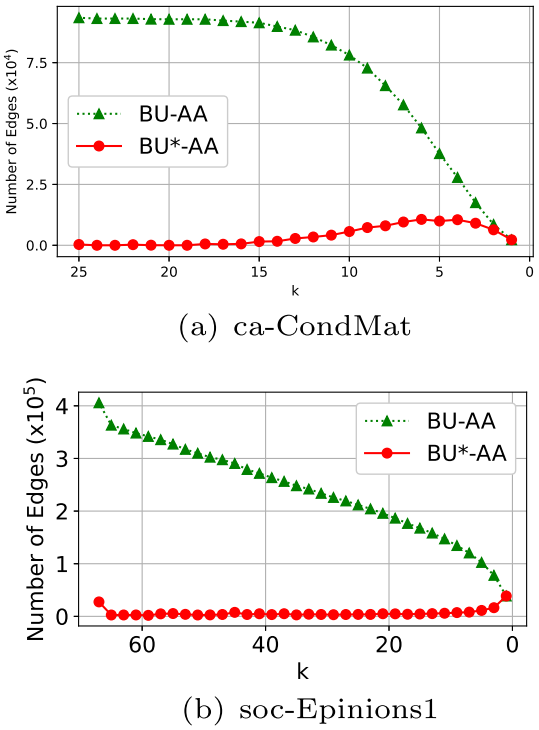
Fig. 16 Comparing BU*-AA with DC-AA on real graphs

increases with m. We can see that BU-AA now runs slower than DC-AA when the degeneracy becomes large, e.g., the degeneracy of PL7_5 and PL7_6 are 705 and 1, 380, respectively. The processing time and memory usage of the six ECo-decomposition algorithms on SSCA graphs are, respectively, shown in Fig. 15c, f, which have similar trends as on the power-law graphs that vary m while fixing n. This is because SSCA graphs as well as these power-law graphs have large degeneracy, e.g., the degeneracy of SSCA12 is 202. From Fig. 15, we can also observe that DC-AA scales almost linearly to large graphs for both the processing time and the memory usage.

7.1.2 Comparing BU*-AA with DC-AA

In this subsection, we compare the efficiency of our simpler approach BU*-AA with the near-optimal approach DC-AA. The results on real graphs are illustrated in Fig. 16a. We can see that BU*-AA consistently runs faster than DC-AA. For example, the processing time of BU*-AA on D13 and D14 is, respectively, 29 min and 1.5hrs, while the processing time of DC-AA on D13 and D14 is, respectively, 78 min and 3.3hrs. Recall that BU*-AA is optimized upon BU-AA by k-core reduction, and BU-AA runs much slower than DC-AA (e.g., on D13) as shown in Fig. 14. This demonstrates that our k-core reduction significantly improves the efficiency of BU-AA. To get more insights, we also show the size of the graph that is input to KECC by BU*-AA and BU-AA, for different k values. The results on ca-CondMat and soc-Epinions1 are reported in Fig. 17. We can see that the input graph to KECC by BU*-AA is significantly smaller than that by BU-AA, especially for large k values; note that, for $k = 1$, they are the same, as one would expect. This demonstrates the effectiveness of our k-core reduction in BU*-AA.

The memory usage of BU*-AA and DC-AA on real graphs is shown in Fig. 16b. We can see that BU*-AA uses slightly less memory than DC-AA, though the improvement is not significant; this conforms with out theoretical analysis that the space complexity of BU*-AA is $2m + \mathcal{O}(n)$, while the space complexity of DC-AA is $2m + \mathcal{O}(n \log n)$.



(a) ca-CondMat

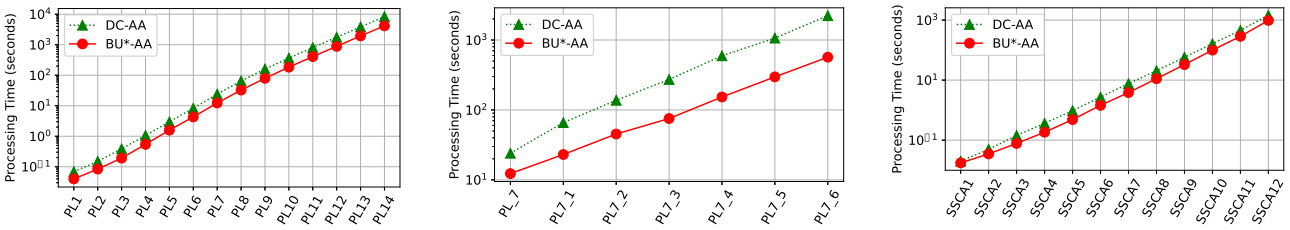
(b) soc-Epinions1

Fig. 17 The size of the input graph to KECC by BU*-AA and BU-AA

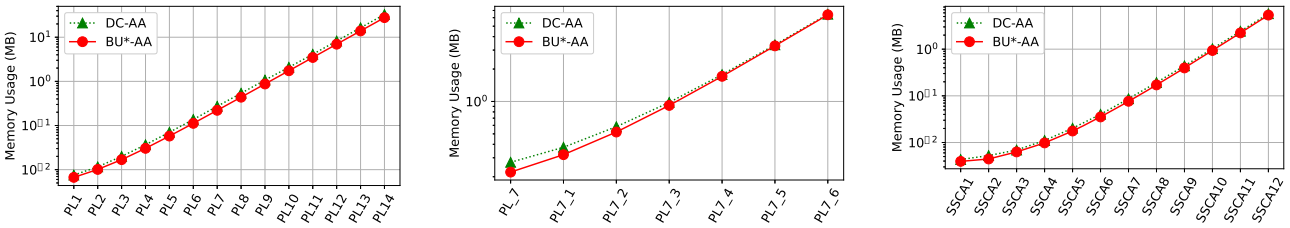
The processing time and memory usage of BU*-AA and DC-AA on synthetic graphs are reported in Fig. 18. The results are similar to that in Fig. 16 for real graphs. That is, BU*-AA runs faster and uses less memory than DC-AA.

7.2 Results for k-ECC computation

In this subsection, we evaluate our space-optimized algorithm KECC-AA for k-ECC computation. We first compare KECC-AA with the linked list-based counterpart KECC that is proposed in [11]. The results on real graphs for $k = 8$ are shown in Fig. 19. We can observe that KECC-AA significantly reduces the memory usage compared with KECC. For example, KECC consumes 78 GB and 119 GB memory, respectively, for processing D13 and D14, while KECC-AA only consumes 11 GB and 17 GB memory for these two graphs. It is also interesting to see that KECC-AA is slightly

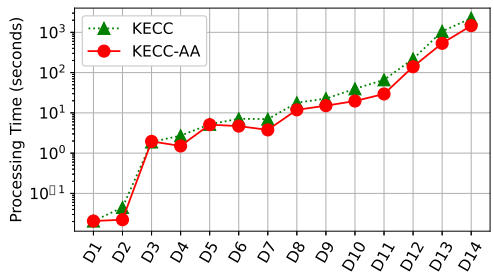


(a) Processing time on power-law graphs (vary n and m, fix average degree)
(b) Processing time on power-law graphs (vary m, fix n)
(c) Processing time on SSCA graphs

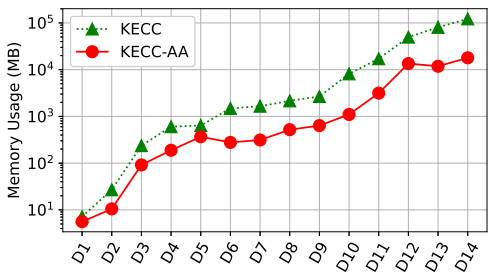


(d) Memory usage on power-law graphs (vary n and m, fix average degree)
(e) Memory usage on power-law graphs (vary m, fix n)
(f) Memory usage on SSCA graphs

Fig. 18 Comparing BU*-AA with DC-AA on synthetic graphs



(a) Processing time (k-ECC computation)



(b) Memory usage (k-ECC computation)

Fig. 19 Results of k-ECC on real graphs

faster than KECC. This is because KECC-AA benefits from increased cache hit rate by using adjacency array-based graph representation.

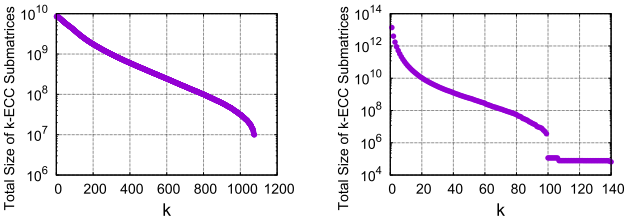
We also compare KECC-AA with the k-ECC computation algorithm in NetworkX, a popular Python module for graph analytics. The results on the two smallest real graphs D1 and D2 for $k = 8$ are shown in Table 2; we do not test NetworkX on larger graphs as it is too slow. We can see that KECC-AA significantly outperforms NetworkX for k-ECC computation, e.g., on D2, KECC-AA is more than 60,000 times faster and consumes 32 times less memory than NetworkX. Although there are factors of programming language difference (i.e., C++ vs. Python), it is clear that KECC-AA has significant advantages over the implementation in NetworkX. It will be an interesting future work to implement KECC-AA in NetworkX.

7.3 Applications

In this subsection, we illustrate applying our ECo-decomposition algorithms in applications. Firstly, our algorithms directly speed up the index construction for steiner component search studied in [7, 25], which use the hierarchy tree as an index structure for efficiently processing online queries. Secondly, our algorithms can facilitate matrix completability analysis, where matrix completion is typically used for recommendation [14]. Specifically, a matrix can be represented as a bipartite graph $G = (U \cup L, E)$ with $U \cap L = \emptyset$ and $E \subseteq U \times L$. Each row of the matrix corresponds to a vertex of U, each column corresponds to a vertex of L, and each non-zero entry at position (i, j) corresponds to an undirected edge between $i \in U$ and $j \in L$. The problem of matrix completion is to predicate values for the entries of the matrix that currently have value 0 (i.e., with value missing). It has been shown in [14] that the higher

Table 2 Compare KECC-AA with NetworkX ($k = 8$)

Dataset	NetworkX		KECC-AA	
	Time (s)	Memory (MB)	Time (s)	Memory
D1	768.89	164.66	0.021	5.73
D2	1412.99	772.74	0.022	23.16



(a) Netflix (b) Amazon_reviews

Fig. 20 Matrix completability analysis

the edge connectivity of the corresponding bipartite graph, the more accurate the low-rank matrix completion. Thus, the higher the value of k such that i and j are contained in the same k-ECC, the more accurate the predicated value of the (i, j)th entry of the matrix. The hierarchy tree constructed by our algorithms can be used to efficiently obtain the largest k such that i and j are contained in the same k-ECC and thus to estimate the accuracy of the matrix completion for the (i, j)th entry. Also, the hierarchy tree can be used to efficiently retrieve the submatrices, whose corresponding bipartite graphs are k-edge connected, to run the matrix completion algorithm and can be used to provide a guide on choosing the appropriate k. For example, Fig. 20a, b shows the total size of the submatrices whose corresponding bipartite graphs are k-edge connected, for datasets Netflix and Amazon_reviews; here, the size of a submatrix is $\#rows \times \#columns$. Netflix [10] has $|U| = 480{,}189$, $|L| = 17{,}770$, $|E| = 100{,}480{,}507$, $k_{\max} = 1{,}076$, and Amazon_reviews [11] has $|U| = 6{,}643{,}669$, $|L| = 2{,}441{,}053$, $|E| = 29{,}928{,}296$, $k_{\max} = 140$. We can see that Netflix is much denser than Amazon_reviews and can be completed more accurately than Amazon_reviews. In particular, the total size of the submatrices whose corresponding bipartite graphs are 200-edge connected is more than 10% of the entire matrix size for Netflix, while there is no such submatrix for Amazon_reviews.

[10] https://www.kaggle.com/netflix-inc/netflix-prize-data.

[11] http://snap.stanford.edu/data/web-Amazon-links.html.

8 Related work

Besides the existing works on ECo-decomposition as discussed in Sects. 1 and 3, we categorize other related works as follows.

k-ECC

Computation. In the literature, there are three approaches for computing all k-ECCs of a graph for a given k: cut-based approach [31, 40, 43], decomposition-based approach [11], and randomized approach [3]. In this paper, we adopted the decomposition-based approach [11] for k-ECC computation—which is the state of the art—and further optimized its memory usage.

Edge Connectivity Computation. Computing the edge connectivity between two vertices has been studied in graph theory [21], which is achieved by maximum flow techniques [17]. The state-of-the-art algorithms compute the maximum flow exactly in $\mathcal{O}(n \times m)$ time [30] and approximately in almost linear time [26, 35]. Index structures have also been developed to efficiently process vertex-to-vertex edge connectivity queries [1, 23]. However, steiner connectivity as computed in this paper, which measures the connectivity in a *subgraph*, is different from edge connectivity as computed in [1, 23], which measures the connectivity in the *input graph*. Thus, these techniques cannot be applied. Moreover, it is worth mentioning that none of our algorithms involve maximum flow computation.

Cohesive Subgraph Computation. Extracting cohesive subgraphs from a large graph has also been extensively studied in the literature (see [9] for a recent survey). Here, the cohesiveness of a subgraph usually is measured by the minimum degree (aka, k-core) [34], the average degree (aka, edge density) [8, 12, 22], the minimum number of triangles each edge participates in (aka, k-truss) [16, 32], and the vertex connectivity [38]. For some of the measures, the cohesive subgraphs for different cohesiveness values also form hierarchical structures and efficient algorithms have been proposed to construct these hierarchical structures, e.g., core decomposition [15], truss decomposition and its higher-order variants [33], and density-friendly graph decomposition [18, 37]. However, due to inherently different problem natures, these techniques are inapplicable to computing ECo-decomposition of a graph.

Community Search. The problem of community search, which aims to find a densely connected subgraph containing user-given query vertices, has received a lot of attention recently. Edge connectivity is used as one of the density measures [7, 24, 25]; the techniques proposed in this paper can be used to speed up their index construction. Other density measures include minimum degree, average degree, minimum number of triangles each edge participates in, etc. A

comprehensive survey of community search with respect to the different density measures can be found in [19].

Modularity, the classic density measure that was originally designed for partitioning a graph into communities (referred to as the community detection problem in the literature) was recently extended to the community search problem [27]. Efficient techniques have been proposed in [41] for finding a good community whose number of vertices falls in a user-given range. With the success of deep learning in many domains, graph neural networks have also been recently used for solving the community search problem [13].

9 Conclusion

In this paper, we proposed a near-optimal algorithm DC-AA for constructing the hierarchy tree of k-ECCs for all possible k values. DC-AA has both a lower time complexity and a lower space complexity compared with the state-of-the-art approaches TD and BU. Extensive experimental results on large graphs demonstrate that DC-AA outperforms TD and BU by up to 28 times in terms of running time and by up to 8 times regarding memory usage. As a result, DC-AA makes it possible to process billion-scale graphs in the main memory of a commodity machine. As a by-product, we also significantly reduced the memory usage of the state-of-the-art k-ECC computation algorithm. Moreover, we also proposed a simpler algorithm BU*-AA that is much easier to implement and runs faster than DC-AA in practice.

Acknowledgements This work was supported by the Australian Research Council Funding of FT180100256 and DP220103731.

Funding Open Access funding enabled and organized by CAUL and its Member Institutions.

References

1. Aggarwal, C.C., Xie, Y., Philip, S.Y.: Gconnect: a connectivity index for massive disk-resident graphs. PVLDB **2**(1), 862–873 (2009)
2. Agrawal, R., Rajagopalan, S., Srikant, R., Xu, Y.: Mining newsgroups using networks arising from social behavior. In: Proceedings of WWW'03, pp. 529–535 (2003)
3. Akiba, T., Iwata, Y., Yoshida, Y.: Linear-time enumeration of maximal k-edge-connected subgraphs in large networks by random contraction. In: Proceedings of CIKM'13, pp. 909–918 (2013)
4. Batagelj, V., Zaversnik, M.: An o(m) algorithm for cores decomposition of networks. CoRR arXiv:cs.DS/0310049 (2003)
5. Benczúr, A.A., Karger, D.R.: Randomized approximation schemes for cuts and flows in capacitated graphs. CoRR arXiv:cs.DS/0207078 (2002)
6. Carmi, S., Havlin, S., Kirkpatrick, S., Shavitt, Y., Shir, E.: A model of internet topology using k-shell decomposition. Proc. Natl. Acad. Sci. USA **104**(27), 11150–11154 (2007)
7. Chang, L., Lin, X., Qin, L., Yu, J.X., Zhang, W.: Index-based optimal algorithms for computing Steiner components with maximum connectivity. In: Proceedings of SIGMOD'15 (2015)
8. Chang, L., Qiao, M.: Deconstruct densest subgraphs. In: Proceedings of WWW'20, pp. 2747–2753 (2020)
9. Chang, L., Qin, L.: Cohesive Subgraph Computation over Large Sparse Graphs. Springer Series in the Data Sciences (2018)
10. Chang, L., Wang, Z.: A near-optimal approach to edge connectivity-based hierarchical graph decomposition. Proc. VLDB Endow. **15**(6), 1146–1158 (2022)
11. Chang, L., Yu, J.X., Qin, L., Lin, X., Liu, C., Liang, W.: Efficiently computing k-edge connected components via graph decomposition. In: Proceedings of SIGMOD'13, pp. 205–216 (2013)
12. Charikar, M.,: Greedy approximation algorithms for finding dense components in a graph. In: Proceedings of APPROX'00, pp. 84–95 (2000)
13. Chen, J., Gao, J., Cui, B.: Ics-gnn$^+$: lightweight interactive community search via graph neural network. VLDB J. **32**(2), 447–467 (2023)
14. Cheng, D., Ruchansky, N., Liu, Y.: Matrix completability analysis via graph k-connectivity. In: Proceedings of AISTATS'18, pp. 395–403 (2018)
15. Cheng, J., Ke, Y., Chu, S., Özsu, M.T.: Efficient core decomposition in massive networks. In: Proceedings of ICDE'11, pp. 51–62 (2011)
16. Cohen, J.: Trusses: Cohesive subgraphs for social network analysis. National Security Agency Technical Report, p. 16 (2008)
17. Cormen, T.H., Stein, C., Rivest, R.L., Leiserson, C.E.: Introduction to Algorithms. McGraw-Hill Higher Education, New York (2001)
18. Danisch, M., Hubert Chan, T.-H., Sozio, M.: Large scale density-friendly graph decomposition via convex programming. In: Proceedings of WWW'17, pp. 233–242 (2017)
19. Fang, Y., Xin Huang, L., Qin, Y.Z., Zhang, W., Cheng, R., Lin, X.: A survey of community search over big graphs. VLDB J. **29**(1), 353–392 (2020)
20. Fung, W.S., Hariharan, R., Harvey, N.J.A., Panigrahi, D.: A general framework for graph sparsification. In: Proceedings of STOC'11, pp. 71–80 (2011)
21. Gibbons, A.: Algorithmic Graph Theory. Cambridge University Press, Cambridge (1985)
22. Goldberg, A.V.: Finding a maximum density subgraph. Technical report, Berkeley, CA, USA (1984)
23. Gomory, R.E., Hu, T.C.: Multi-terminal network flows. J. Soc. Ind. Appl. Math. **9**(4), 551–570 (1961)
24. Hu, J., Wu, X., Cheng, R., Luo, S., Fang, Y.: Querying minimal Steiner maximum-connected subgraphs in large graphs. In: Proceedings of CIKM'16, pp. 1241–1250 (2016)
25. Hu, J., Wu, X., Cheng, R., Luo, S., Fang, Y.: On minimal Steiner maximum-connected subgraph queries. IEEE Trans. Knowl. Data Eng. **29**(11), 2455–2469 (2017)
26. Kelner, J.A., Lee, Y.T., Orecchia, L., Sidford, A.: An almost-linear-time algorithm for approximate max flow in undirected graphs, and

its multicommodity generalizations. In: Proceedings of SODA'13 (2013)
27. Kim, J., Luo, S., Cong, G., Yu, W.: DMCS: Density modularity based community search. In: Proceedings of SIGMOD'22, pp. 889–903 (2022)
28. Matula, D.W., Beck, L.L.: Smallest-last ordering and clustering and graph coloring algorithms. J. ACM **30**(3), 417–427 (1983)
29. Nguyen, A., Hong, S.-H.: k-core based multi-level graph visualization for scale-free networks. In: Proceedings of PacificVis'17, pp. 21–25 (2017)
30. Orlin, J.B.: Max flows in o(nm) time, or better. In: Proceedings of STOC'13, pp. 765–774 (2013)
31. Papadopoulos, A.N., Lyritsis, A., Manolopoulos, Y.: Skygraph: an algorithm for important subgraph discovery in relational graphs. Data Min. Knowl. Discov., 17(1), August 2008
32. Saito, K., Yamada, T.: Extracting communities from complex networks by the k-dense method. In: Proceedings of ICDMw'06, pp. 300–304 (2006)
33. Sariyüce, A.E., Pinar, A.: Fast hierarchy construction for dense subgraphs. PVLDB **10**(3), 97–108 (2016)
34. Seidman, S.B.: Network structure and minimum degree. Social Networks **5**(3), 269–287 (1983)
35. Sherman, J.: Nearly maximum flows in nearly linear time. In: Proceedings of FOCS'13 (2013)
36. Sorge, M., et al.: The graph parameter hierarchy (2013)
37. Sun, B., Danisch, M., Hubert Chan, T.-H., Sozio, M.: Kclist++: a simple algorithm for finding k-clique densest subgraphs in large graphs. Proc. VLDB Endow. **13**(10), 1628–1640 (2020)
38. Wen, D., Qin, L., Zhang, Y., Chang, L., Chen, L.: Enumerating k-vertex connected components in large graphs. In: Proceedings of ICDE'19, pp. 52–63 (2019)
39. White, D.R., Harary, F.: The cohesiveness of blocks in social networks: node connectivity and conditional density. Sociol. Methodol. **31**, 305–359 (2001)
40. Yan, X., Jasmine Zhou, X., Han, J.: Mining closed relational graphs with connectivity constraints. In: Proceedings of KDD'05 (2005)
41. Yao, K., Chang, L.: Efficient size-bounded community search over large networks. Proc. VLDB Endow. **14**(8), 1441–1453 (2021)
42. Yuan, L., Qin, L., Lin, X., Chang, L., Zhang, W.: I/O efficient ECC graph decomposition via graph reduction. VLDB J. **26**(2), 275–300 (2017)
43. Zhou, R., Liu, C., Yu, J.X., Liang, W., Chen, B., Li, J.: Finding maximal k-edge-connected subgraphs from a large graph. In: Proceedings of EDBT'12 (2012)