A Near-Optimal Approach to Edge Connectivity-Based Hierarchical Graph Decomposition

Lijun Chang
The University of Sydney
Lijun.Chang@sydney.edu.au

Zhiyi Wang
The University of Sydney
zwan9517@uni.sydney.edu.au

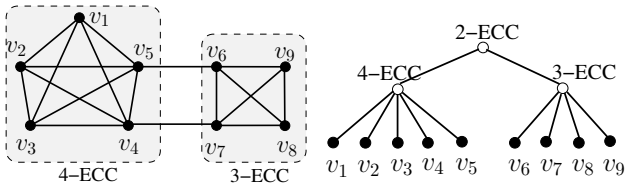
ABSTRACT

Driven by applications in graph analytics, the problem of efficiently computing all k-edge connected components (k-ECCs) of a graph G for *a user-given* k has been extensively and well studied. It is known that the k-ECCs of G for *all possible values of* k form a hierarchical structure. In this paper, we study the problem of efficiently constructing the hierarchy tree for G which compactly encodes the k-ECCs for all possible k values in space linear to the number of vertices n. All existing approaches construct the hierarchy tree in $O\big(\delta(G) \times \mathsf{T}_{\mathsf{KECC}}(G)\big)$ time, 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. To improve the time complexity, we propose a divide-and-conquer approach running in $O\big((\log \delta(G)) \times \mathsf{T}_{\mathsf{KECC}}(G)\big)$ time, which is optimal up to a logarithmic factor. However, a straightforward implementation of our algorithm would result in a space complexity of $O((m+n)\log \delta(G))$. As main memory also becomes a scarce resource when processing large-scale graphs, we further propose techniques to optimize the space complexity to $2m + O(n \log \delta(G))$, where m is the number of edges in G. Extensive experiments on large real graphs and synthetic graphs demonstrate that our approach outperforms 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. As a by-product, we also improve the space complexity of computing all k-ECCs for a specific k to $2m + O(n)$.

PVLDB Reference Format:
Lijun Chang and Zhiyi Wang. A Near-Optimal Approach to Edge Connectivity-Based Hierarchical Graph Decomposition. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

1 INTRODUCTION

Graphs have been widely used to model the relationships among entities in real-world applications — such as social networks, collaboration networks, communication networks, E-commerce networks, web search, and biology — where entities are represented by vertices and relationships are represented by edges. With the proliferation of graph data, one of the fundamental problems in graph analytics 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 [4, 9, 34, 36]. A graph is *k-edge connected*, if it remains connected after removing any set of $k-1$ edges. For example, for the graph in Figure 1(a), the subgraphs induced by vertices $\{v_1, \ldots, v_5\}$ and $\{v_6, \ldots, v_9\}$ are the two 3-ECCs, while the former is also a 4-ECC. Computing k-ECCs has many applications, such as discovering cohesive blocks (communities) in social networks (*e.g.*, Facebook) [33], identifying closely related entities for social behavior mining [3], measuring robustness of communication networks [9], and matrix completability analysis [11].



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

Figure 1: A toy graph and its ECo-decomposition

Specifying the appropriate k value for an application is however not trivial and usually requires a trial-and-error process. Moreover, different applications may specify different k values. Thus, it is essential to pre-compute a data structure, such that k-ECCs for any given k can be efficiently retrieved from the data structure. It is known that the k-ECCs for all possible values of k form a hierarchical structure [35], as the k-ECCs for a specific k are disjoint and each k-ECC is entirely contained in a $(k-1)$-ECC [7]. For example, Figure 1(b) depicts the *hierarchy tree* $\mathcal{T}$ for the k-ECCs of the graph G in Figure 1(a), where leaf nodes are vertices of G and non-leaf nodes correspond to k-ECCs of G. With the constructed tree $\mathcal{T}$, the set of k-ECCs for any k can be extracted from $\mathcal{T}$ in time linear to the size of the k-ECCs. Thus, it becomes a problem of efficiently constructing the hierarchy tree for k-ECCs of all possible k values. We term the problem as *Edge Connectivity-based hierarchical graph decomposition*, abbreviated as ECo-*decomposition*.

Besides inheriting all the above applications, computing ECo-decomposition (*i.e.*, the hierarchy tree) also has a wide range of other applications as follows.

- *Hierarchical Organization and Visualization of Graphs.* ECo-decomposition constructs a hierarchical organization of a graph. It can facilitate graph-topology analysis [6], and assist users to visualize a graph in a multi-granularity manner [23], *i.e.*, zoom in and zoom out based on the edge connectivities of subgraphs.
- *Graph Sparsification.* ECo-decomposition efficiently computes the steiner connectivity for all edges (see Section 4.1). It is shown in [5, 16] that independently sampling edges according to their steiner connectivities can sparsify a graph (*i.e.*, reduce the number of edges) while preserving the values of all cuts with a small multiplicative error.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

- *Steiner Component Search.* ECo-decomposition is also an inherent preprocessing step towards efficient online steiner component search [7, 20], which is the problem of computing the subgraph with the maximum edge connectivity for a user-given set of query vertices [7].

The state-of-the-art approaches compute the ECo-decomposition (*i.e.*, construct the hierarchy tree $\mathcal{T}$) either in a top-down manner [7] or a bottom-up manner [35]. The top-down approach ECo-TD constructs the hierarchy tree by computing k-ECCs of G for all possible k values in increasing order [7], while the bottom-up approach ECo-BU computes k-ECCs of G for all possible k values in decreasing order [35]. Computation sharing techniques are exploited in ECo-TD and ECo-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 ECo-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]. Nevertheless, the worst-case time complexities of ECo-TD and ECo-BU are still $O\big(\delta(G) \times \mathsf{T}_{\mathsf{KECC}}(G)\big)$, 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 which is equal to the maximum value among the minimum vertex degrees of all subgraphs of G [22]. It is interesting to observe that this time complexity 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. In this paper, we separate the computation into two parts: we first compute the steiner connectivity for all edges of G, and then construct the hierarchy tree $\mathcal{T}$ based on the computed steiner connectivities. The *steiner connectivity* of an edge (u, v), denoted as $sc(u, v)$, is the largest k such that a k-ECC of G contains (u, v). We show in the paper that the hierarchy tree of the ECo-decomposition can be constructed in $O(m)$ time given the steiner connectivities of all edges of G, where m is the number of edges of G. As a result, 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 ECo-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 sub-problems, 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 procedure is to efficiently divide a search problem E_L^H into two: E' and E''. We prove 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 $E'' = E_L^H \backslash E'$. In addition, computation sharing techniques are exploited to bound the time complexity of ECo-DC by $O\big((\log \delta(G)) \times \mathsf{T}_{\mathsf{KECC}}(G)\big)$.

ECo-DC is optimal up to a logarithmic factor in terms of time complexity, since the time complexity of an ECo-decomposition algorithm is clearly lower bounded by $\mathsf{T}_{\mathsf{KECC}}(G)$. However, a naive implementation of ECo-DC would result in a space complexity of $O\big((n+m) \log \delta(G)\big)$ which is infeasible for large graphs. We first show that the space complexity can be reduced to $O(m+n \log \delta(G))$. Although this is much lower than the naive implementation, it is still too high to be applied to billion-scale graphs due to running out-of-memory, as the constant hidden by the big-O notation is large. In view of this, we further propose techniques to reduce the space complexity to $2m + O(n \log \delta(G))$ by explicitly bounding the constant on m by 2, while not increasing the time complexity; our space-optimized approach is denoted as ECo-DC-AA.

Extensive empirical studies on large graphs demonstrate that our approach ECo-DC-AA outperforms the state-of-the-art approaches ECo-TD and ECo-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, ECo-DC-AA finishes in 78 minutes by consuming 15GB memory, while ECo-TD and ECo-BU (as well as ECo-DC) run out-of-memory on a machine with 128GB memory; on the other hand, our space-optimized versions of ECo-TD and ECo-BU finish in 13.9 and 36.8 hours, respectively.

Our main contributions are summarized as follows.

- We propose a near-optimal approach to ECo-decomposition, which reduces the time complexity from $O(\delta(G) \times \mathsf{T}_{\mathsf{KECC}}(G))$ to $O((\log \delta(G)) \times \mathsf{T}_{\mathsf{KECC}}(G))$.
- We propose techniques to reduce the space complexity of our approach from $O\big((n+m) \log \delta(G)\big)$ to $2m + O(n \log \delta(G))$, such that billion-scale graphs can be processed in the main memory of a commodity machine.
- As a by-product, we significantly reduce the memory usage of the state-of-the-art k-ECC computation algorithm proposed in [9]. Moreover, our space optimization techniques can be generally applied to other graph algorithms.
- We conduct extensive empirical studies on large real and synthetic graphs to evaluate the efficiency of our approaches.

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

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.

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.

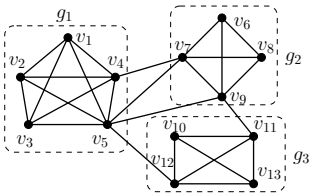


Figure 2: An example graph

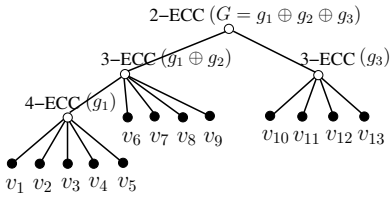


Figure 3: Hierarchy tree $\mathcal{T}$

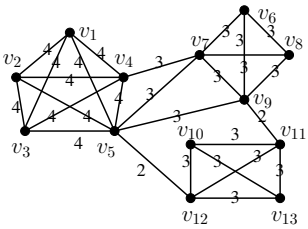


Figure 4: Steiner connectivities

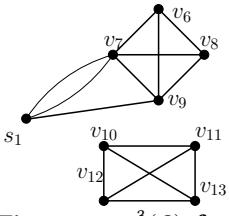


Figure 5: $\mathtt{GS}_3^3(G)$ for the graph in Figure 2

Definition 2.1: (k-edge Connected Component [9]) 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 Figure 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 Figure 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 Figure 3 corresponds to the 3-ECC $g_1 \oplus g_2$ in Figure 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 $\mathtt{KECC}(g, k)$. While any of the algorithms in [4, 9, 36] can be used to implement $\mathtt{KECC}(g, k)$, we implement the state-of-the-art algorithm in [9] in our experiments, and use $\mathsf{T}_{\mathtt{KECC}}(G)$ to denote the time complexity of $\mathtt{KECC}(g, k)$ when G is taken as the input graph.

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 [35].

A Top-Down Approach: ECo-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 Algorithm 1, denoted by ECo-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).

Algorithm 1: ECo-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      for each 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     for each 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');
```

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 $\phi_{k+1}(g)$

Algorithm 2: ECo-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      for each 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;
```

(Lines 9–10); (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).

A Bottom-Up Approach: ECo-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 [35]. The pseudocode is shown in Algorithm 2, denoted ECo-BU.

Descriptions of Algorithm 2 as well as running examples of ECo-TD and ECo-BU can be found in Appendix A.

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

Lemma 3.1: *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 [22]. Then, we have $k_{\max}(G) \leq \delta(G)$.*

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 $\delta(G)$, and the time complexity of ECo-TD is $O(\delta(G) \times \mathsf{T}_{\mathsf{KECC}}(G))$.[1] Note that, the time complexity analysis of ECo-TD is tight: for example, consider an input graph G that itself is $\delta(G)$-edge connected.

Following Lemma 3.1, the upper bound $\overline{k_{\max}}(G)$ can be set as $\delta(G)$ at Line 2 of Algorithm 2. Thus, the time complexity of ECo-BU is $O(\delta(G) \times \mathsf{T}_{\mathsf{KECC}}(G))$,[2] as the degeneracy of G can be computed in $O(m)$ time [22]. Note that, the time complexity analysis of ECo-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.

It is interesting to observe that the time complexity of ECo-TD and ECo-BU is the same as the straightforward approach that first independently computes k-ECCs of G for all possible k values and then constructs the hierarchy tree based on the computed k-ECCs.

The degeneracy $\delta(G)$, although can be bounded by $O(\sqrt{m})$ in the worst case [30], may still be large, especially for large graphs. For example, $\delta(G)$ is more than 2,000 for the largest graphs tested in our experiments (see Table 1 in Section 6). As a result, ECo-BU and ECo-TD are taking excessively long time for processing large graphs due to their high time complexity of $O(\delta(G) \times \mathsf{T}_{\mathsf{KECC}}(G))$, not to mention their high space complexity (see Section 5).

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 $O((\log \delta(G)) \times \mathsf{T}_{\mathsf{KECC}}(G))$ time. To achieve this, we will need to avoid the explicit computation and enumeration of k-ECCs for all possible k values which would take $O(\delta(G) \times \mathsf{T}_{\mathsf{KECC}}(G))$ 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;
```

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

4.1 Computing Steiner Connectivities

Definition 4.1: (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 Figure 4, the steiner connectivity of each edge is computed as shown on the edge, *e.g.*, $sc(v_1, v_4) = 4$. 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 $\mathsf{GS}_{k_1}^{k_2}(\cdot)$. We first introduce a graph shrink operator $\mathsf{GS}_{k_1}^{k_2}(\cdot)$ for $k_1 \leq k_2$. Given a graph G, the result of $\mathsf{GS}_{k_1}^{k_2}(G)$ 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 $\mathsf{GS}_{k_1}^{k_2}(\cdot)$ may have parallel edges. For example, $\mathsf{GS}_3^3(G)$ for the graph G in Figure 2 is shown in Figure 5 which is obtained

[1] Although the time complexity of ECo-TD is analyzed to be $O(\alpha(G) \times \mathsf{T}_{\mathsf{KECC}}(G))$ in [7] where $\alpha(G)$ is the arboricity of G, this is the same as $O(\delta(G) \times \mathsf{T}_{\mathsf{KECC}}(G))$ since $\alpha(G) \leq \delta(G) \leq 2\alpha(G) - 1$ [30].

[2] It is worth pointing out that the original algorithm in [35] is designed for I/O-efficient settings, and its time complexity cannot be bounded by $O(\delta(G) \times \mathsf{T}_{\mathsf{KECC}}(G))$ as the upper bound $\overline{k_{\max}}(G)$ is set as the maximum degree of G in [35].

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 Figure 5.

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.*

Secondly, the steiner connectivity for all edges of $\mathtt{GS}_k^k(G)$ is k. For example, all edges in Figure 5 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.*

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 $\mathtt{GS}_{k_3}^{k_4}\big(\mathtt{GS}_{k_1}^{k_2}(G)\big) = \mathtt{GS}_{\max\{k_1,k_3\}}^{\min\{k_2,k_4\}}(G)$.*

Our Divide-and-Conquer Approach: ECo-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 $O\big(\delta(G) \times \mathsf{T}_{\mathsf{KECC}}(G)\big)$ 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 $k_1 \le k \le k_2$.

Algorithm 3: ECo-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      for each 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
         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);
```

The pseudocode of our approach is shown in Algorithm 3, denoted ECo-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 and will be discussed in Section 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 $L < M \le H$ (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 4.1: *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.*

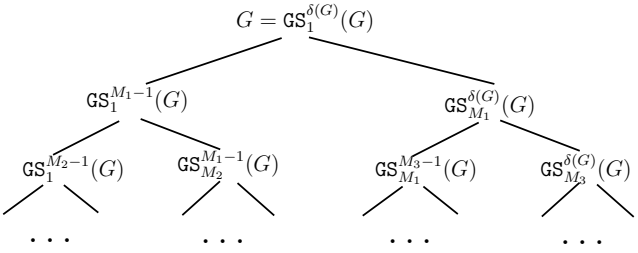


Figure 6: Recursion tree

Based on Lemma 4.1, the recursion tree of invoking Compute-DC with input $(G, 1, \delta(G))$ is as shown in Figure 6.

The correctness and time complexity of Algorithm 3 are proved by the two theorems below.

Theorem 4.1: *Algorithm 3 correctly computes the steiner connectivity for all edges of G.*

Theorem 4.2: *The time complexity of Algorithm 3 is $O(h \times \mathsf{T}_{\mathsf{KECC}}(G))$, where h is the height of the recursion tree in Figure 6.*

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 $L < M \le H$. 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 $O(\delta(G) \times \mathsf{T}_{\mathsf{KECC}}(G))$ on the basis of Theorem 4.2. 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 $O\big((\log \delta(G)) \times \mathsf{T}_{\mathsf{KECC}}(G)\big)$.

Theorem 4.3: *By setting $M = \lceil \frac{L+H}{2} \rceil$, the time complexity of Algorithm 3 is $O\big((\log \delta(G)) \times \mathsf{T}_{\mathsf{KECC}}(G)\big)$.*

Following the above theorem, we set $M = \lceil \frac{L+H}{2} \rceil$ in Algorithm 3. The time complexity of ECo-DC, which is $O\big((\log \delta(G)) \times \mathsf{T}_{\mathsf{KECC}}(G)\big)$, 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.

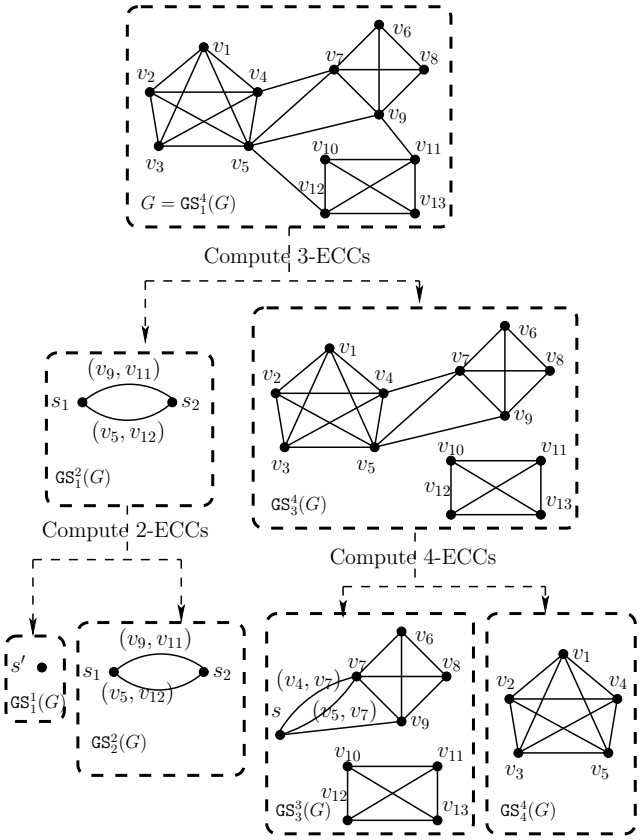


Figure 7: Running example of ECo-DC

Example 4.1: Here, we apply ECo-DC on the graph G in Figure 2 as an example. Figure 7 indicates the whole running process of ECo-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 Figure 7. 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 Figure 7. 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 Figure 7. 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}), respectively. 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. □

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 $sc(u, v)$.

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  for each vertex u ∈ V(G) do Insert a singleton node u into T;
3  for each 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;
```

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 $O(n)$ time by tracing the parent pointers starting from node u in the tree, and then the total time complexity of Algorithm 4 would be $O(n \times m)$. This can be improved to $O(m)$ 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*) [14]. There are two operations supported by the data structure $\mathcal{D}$: find the set that contains a specific element; 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 [14] [3] and sorting the edges at Line 3 can be achieved in linear time by counting sort [14], the time complexity of Algorithm 4 is $O(m)$.

5 OPTIMIZING THE SPACE USAGE

A straightforward implementation of Algorithm 3 would result in a space complexity of $O((m+n)\log\delta(G))$, *i.e.*, each level of the recursion tree of Figure 6 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 ECo-DC. We first in Section 5.1 discuss how to implement ECo-DC in $O(m+n\log\delta(G))$ space by using doubly-linked list-based graph representation, where the constant hidden by the big-O notation is large. Then, in Section 5.2 we further optimize the space usage of ECo-DC by using adjacency array-based graph representation and other nontrivial optimizations; this results in our space-efficient algorithm ECo-DC-AA that has space complexity $2m+O(n\log\delta(G))$. As a result, we can process billion-scale graphs with an ordinary PC. For example, experiments in Section 6 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 15GB and 24GB main memory. In contrast, the linked list-based algorithms run out-of-memory even with 128GB memory.

5.1 Doubly-Linked List-based Implementation

In this subsection, we discuss how to implement ECo-DC by using doubly-linked list-based graph representation, which is also the representation used by the state-of-the-art KECC algorithm [9] and the two state-of-the-art ECo-decomposition algorithms [7, 35]. 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 [9] — 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 ECo-TD also uses the vertex removal operation (see Lines 11–13 of Algorithm 1), and ECo-BU uses the vertex contraction operation (see Line 8 of Algorithm 2).

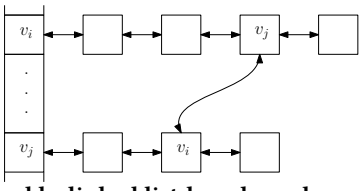


Figure 8: Doubly-linked list-based graph representation

Recall that, the linked list-based graph representation stores the adjacent edges of each vertex in a linked list [14]. For example, Figure 8 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, ECo-DC (*i.e.*, Algorithm 3) can be implemented fairly easily. Specifically, to construct $g_1 = \mathtt{GS}_L^{M-1}(g)$ and $g_2 = \mathtt{GS}_M^H(g) = \phi_M(g)$ from $g = \mathtt{GS}_L^H(G)$ 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, ECo-DC has a space complexity of $O(m+n\log\delta(G))$, by noting that it traverses the recursion tree of Figure 6 in a depth-first manner.

5.2 Adjacency Array-based Implementations

Although the space complexity of ECo-DC has been reduced from $O((m+n)\log\delta(G))$ to $O(m+n\log\delta(G))$ in Section 5.1, this is still too high to be applied to large graphs (see our experimental results in Section 6) as the constant hidden by the big-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

[3] To be more precise, the amortized time complexity of each operation on $\mathcal{D}$ is the inverse of the Ackermann function of n [14]. As this function grows very slowly and is bounded by 4 for all practical values of n, we consider it as a constant.

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 [9]. 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 $2m + O(n \log \delta(G))$, 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.

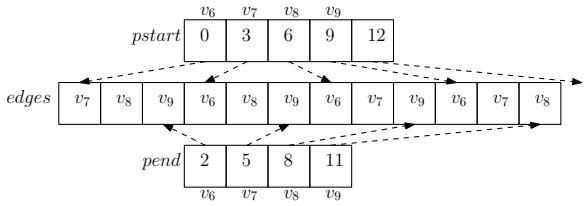


Figure 9: Adjacency array-based graph representation

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, neighbours) 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 9 demonstrates such a representation for the subgraph g_2 of Figure 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 $2m + O(n \log \delta(G))$, 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 $O(n)$.

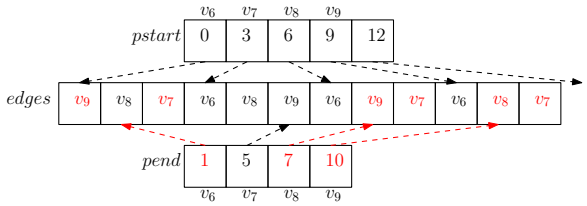


Figure 10: After removing vertex v_7

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* $2m + O(n \log \delta(G))$. 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 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 Figure 9. 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 Figure 9 is shown in Figure 10; 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 Section 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, Figure 11(a) shows the values of *sv_next* and *sv_last* for the graph of Figure 9; 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 not be used. Figure 11(b) 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 $O(n)$ 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

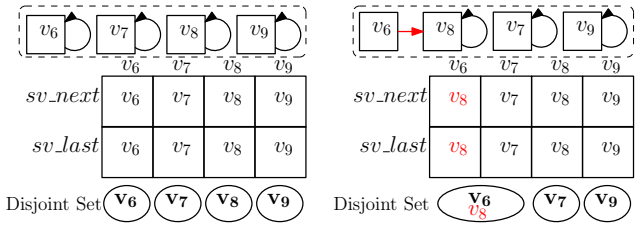


Figure 11: Example of contracting v_6 and v_8

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 Section 4.2. The last row of Figure 11 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 Figure 11(b) 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 ECo-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 $2m + O(n)$.

With KECC-AA, we are now ready to present our space-optimized version of ECo-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 Figure 6) 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 $O(n)$, to implement the contraction operation.

The pseudocode of the adjacency array-based implementation of ECo-DC is illustrated in Algorithm 5, denoted by ECo-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).

Algorithm 5: ECo-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      for each 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     for each 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 reason of interleaving is that explicitly storing the steiner connectivities of all edges would increase the space consumption by at least $2m + O(n)$, 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 (Figure 6), and thus the steiner connectivities are computed in non-increasing order. Note that, *we also exploit this interleaving to reduce the memory consumption for* ECo-DC*,* ECo-TD *and* ECo-BU *in our experiments.*

The correctness of ECo-DC-AA directly follows from the correctness of ECo-DC and the discussions in the above two paragraphs, and the time complexity of ECo-DC-AA remains the same as ECo-DC since our adjacency array-based implementation does not increase the time complexity of vertex removal and contraction. The space complexity of ECo-DC-AA becomes $2m + O(n \log \delta(G))$, as it conducts a depth-first traversal of the recursion tree (Figure 6) and each level of the recursion tree only requires a local data structure of size $O(n)$.

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

6 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:

- ECo-TD (Algorithm 1): the existing top-down approach proposed in [7] that uses the doubly-linked list-based graph representation.
- ECo-BU (Algorithm 2): an adaptation of the existing bottom-up approach proposed in [35] that uses the doubly-linked list-based graph representation.
- ECo-DC (Algorithm 3): our near-optimal approach (Algorithm 3) that uses the doubly-linked list-based graph representation and has a space complexity of $O(m + n \log \delta(G))$.

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

Table 1: Statistics of graphs ($\bar{d}$: average degree, δ: degeneracy)

- ECo-TD-AA, ECo-BU-AA and ECo-DC-AA: space-optimized versions of ECo-TD, ECo-BU and ECo-DC by using the adjacency array-based graph representation (Section 5.2).

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

- KECC: the state-of-the-art algorithm proposed in [9] 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 (Section 5.2).

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 128GB 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 [4] and the Laboratory of Web Algorithmics [5]. 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 evaluate the algorithms on power-law graphs that are generated by the graph generator GTGraph [6]. A power-law graph is a random graph in which edges are randomly added such that the degree distribution follows a power-law distribution. Firstly, we generate fourteen 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 are 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 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.

Furthermore, we also generate twelve SSCA graphs to evaluate the algorithms, the results are reported in Appendix B.

[4]http://snap.stanford.edu/
[5]http://law.di.unimi.it/datasets.php
[6]http://www.cse.psu.edu/~madduri/software/GTgraph/

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 [7].

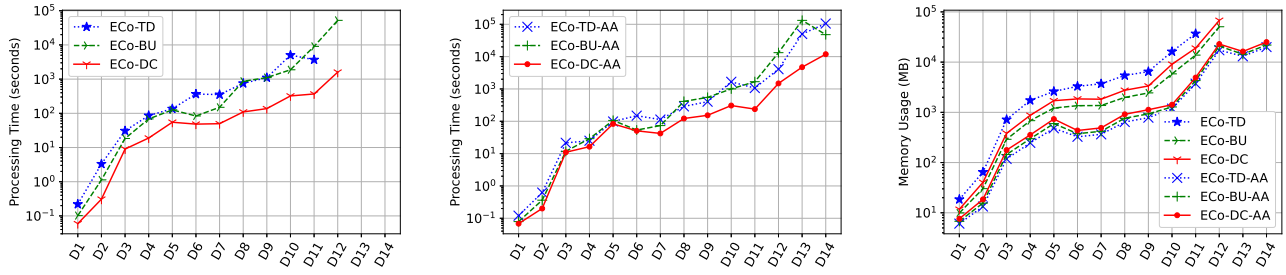
6.1 Results for ECo-decomposition

In this subsection, we evaluate the six ECo-decomposition algorithms 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 Figure 12. For better comparison, we separate the algorithms into two groups: linked list-based algorithms (*i.e.*, ECo-TD, ECo-BU, and ECo-DC), and space-optimized algorithms (*i.e.*, ECo-TD-AA, ECo-BU-AA, and ECo-DC-AA). The processing time of the three linked list-based algorithms is illustrated in Figure 12(a). We can see that our near-optimal approach ECo-DC consistently runs faster than the two state-of-the-art approaches ECo-TD and ECo-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 Figure 12(b), due to their reduced main memory usage. The overall trend is similar to their counterparts in Figure 12(a), *i.e.*, ECo-DC-AA consistently performs the best. When comparing the top-down approach ECo-TD-AA with the bottom-up approach ECo-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 ECo-TD-AA, ECo-BU-AA, and ECo-DC-AA on D13 are respectively 13.9hrs, 36.8hrs and 1.3hrs, while that on D14 are respectively 29.4hrs, 13.3hrs and 3.3hrs.

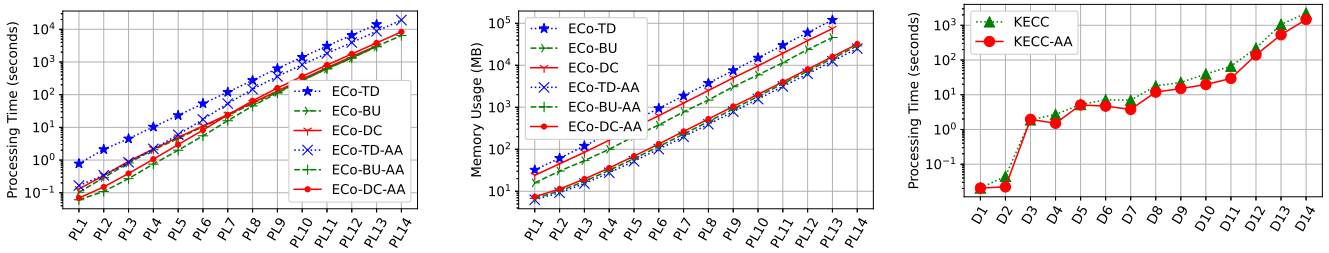
The main memory usage of the six algorithms is demonstrated in Figure 12(c). It is evident that our space-optimized algorithms (ECo-TD-AA, ECo-BU-AA, ECo-DC-AA) consume much less memory than the linked list-based algorithms (ECo-TD, ECo-BU, ECo-DC), where ECo-TD and ECo-BU are the two state-of-the-art approaches. For example, the peak memory usage of our space-optimized algorithms is at most 15GB for D13 and is at most 24GB for D14, while the linked list-based algorithms run out-of-memory even with 128GB memory. There are two things worth mentioning for Figure 12(c). Firstly, it appears that ECo-TD consumes more memory than ECo-DC. This is due to implementation differences, *i.e.*, we used the original implementation of ECo-TD from [7] while our implementations of ECo-BU and ECo-DC slightly optimized the constant on m in the space complexity. We do not optimize the code of ECo-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

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

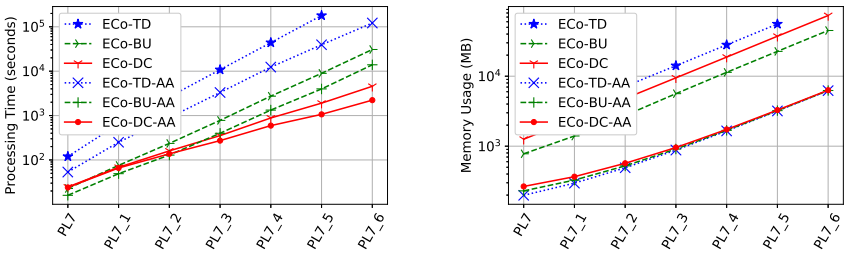


(a) Processing time of linked list-based algorithms (b) Processing time of space-optimized algorithms (c) Memory usage of all algorithms

Figure 12: Results of ECo-decomposition on real graphs (best viewed in color)



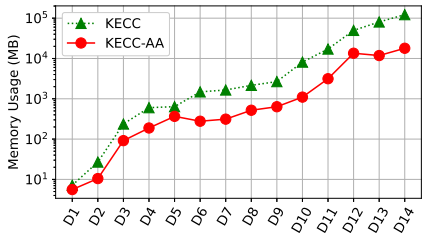
(a) Processing time (vary n and m, fix average degree) (b) Memory usage (vary n and m, fix average degree)



(c) Processing time (vary m, fix n) (d) Memory usage (vary m, fix n)

Figure 13: Results of ECo-decomposition on power-law graphs

(a) Processing time (k-ECC computation)



(b) Memory usage (k-ECC computation)

Figure 14: Results of k-ECC on real graphs

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 Figure 14(b).

Results on Synthetic Graphs. The processing time and memory usage of the six algorithms on power-law graphs are shown in Figure 13. The overall trend is similar to that on real graphs in Figure 12. That is, our divide-and-conquer algorithms ECo-DC and ECo-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 ECo-BU-AA also perform quite well on power-law graphs that have small degeneracy (*i.e.*, at most 25), see Figure 13(a). The results on power-law graphs by varying m and fixing n are shown in Figure 13(c) and Figure 13(d); note that the degeneracy of these graphs also increases with m. We can see that ECo-BU-AA now runs slower than ECo-DC-AA when the degeneracy becomes large, *e.g.*, the degeneracy of PL7_5 and PL7_6 are 705 and 1,380, respectively. From Figure 13, we can also observe that ECo-DC-AA scales almost linearly to large graphs for both the processing time and the memory usage.

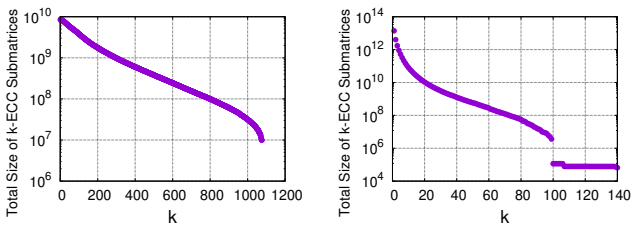
Dataset	NetworkX		KECC-AA	
	Time (s)	Memory (MB)	Time (s)	Memory (MB)
D1	768.89	164.66	0.021	5.73
D2	1412.99	772.74	0.022	23.16

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

6.2 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 [9]. The results on real graphs for $k = 8$ are shown in Figure 14. We can observe that KECC-AA significantly reduces the memory usage compared with KECC. For example, KECC consumes 78GB and 119GB memory respectively for processing D13 and D14, while KECC-AA only consumes 11GB and 17GB memory for these two graphs. It is also interesting to see that KECC-AA is slightly faster than KECC. This is because KECC-AA benefits from increased cache hit-rate by using adjacency array-based graph representation. Similar results are also observed for other k values and for synthetic graphs; they are omitted here due to limit of space.

We also compare KECC-AA with the k-ECC computation algorithm in NetworkX, a popular Python module for graph analytics.



(a) Netflix (b) Amazon_reviews

Figure 15: Matrix completability analysis

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.

6.3 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, 20], 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 [11]. 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 [11] that the higher 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, Figures 15(a) and 15(b) show 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 × #columns. Netflix [8] has $|U| = 480,189$, $|L| = 17,770$, $|E| = 100,480,507$, $k_{\max} = 1,076$, and Amazon_reviews [9] 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.

[8] https://www.kaggle.com/netflix-inc/netflix-prize-data
[9] http://snap.stanford.edu/data/web-Amazon-links.html

7 RELATED WORK

Besides the existing works on ECo-decomposition as discussed in Sections 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 [25, 34, 36], decomposition-based approach [9], and randomized approach [4]. In this paper, we adopted the decomposition-based approach [9] 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 [17], which is achieved by maximum flow techniques [14]. The state-of-the-art algorithms compute the maximum flow exactly in $O(n \times m)$ time [24] and approximately in almost linear time [21, 29]. Index structures have also been developed to efficiently process vertex-to-vertex edge connectivity queries [2, 19]. However, steiner connectivity as computed in this paper, which measures the connectivity in a *subgraph*, is different from edge connectivity as computed in [2, 19], 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 [8] for a recent survey). Here, the cohesiveness of a subgraph usually is measured by the minimum degree (aka, k-core) [28], the average degree (aka, edge density) [10, 18], the minimum number of triangles each edge participates in (aka, k-truss) [13, 26], and the vertex connectivity [32]. 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 [12], truss decomposition and its higher-order variants [27], and density-friendly graph decomposition [15, 31]. However, due to inherently different problem natures, these techniques are inapplicable to computing ECo-decomposition of a graph.

8 CONCLUSION

In this paper, we proposed a near-optimal algorithm ECo-DC-AA for constructing the hierarchy tree of k-ECCs for all possible k values. ECo-DC-AA has both a lower time complexity and a lower space complexity compared with the state-of-the-art approaches ECo-TD and ECo-BU. Extensive experimental results on large graphs demonstrate that ECo-DC-AA outperforms ECo-TD and ECo-BU by up to 28 times in terms of running time and by up to 8 times regarding memory usage. As a result, ECo-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.

REFERENCES

[1] [n.d.]. full version: https://lijunchang.github.io/pdf/ecd-tr.pdf.
[2] Charu C. Aggarwal, Yan Xie, and Philip S. Yu. 2009. GConnect: A Connectivity Index for Massive Disk-resident Graphs. *PVLDB* 2, 1 (2009), 862–873.
[3] Rakesh Agrawal, Sridhar Rajagopalan, Ramakrishnan Srikant, and Yirong Xu. 2003. Mining newsgroups using networks arising from social behavior. In *Proc. of WWW'03*. 529–535.
[4] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Linear-time enumeration of maximal K-edge-connected subgraphs in large networks by random contraction. In *Proc. of CIKM'13*. 909–918.
[5] András A. Benczúr and David R. Karger. 2002. Randomized Approximation Schemes for Cuts and Flows in Capacitated Graphs. *CoRR* cs.DS/0207078 (2002).
[6] Shai Carmi, Shlomo Havlin, Scott Kirkpatrick, Yuval Shavitt, and Eran Shir. 2007. A model of Internet topology using k-shell decomposition. *Proceedings of the National Academy of Sciences of the United States of America* 104, 27 (2007), 11150—-11154.
[7] Lijun Chang, Xuemin Lin, Lu Qin, Jeffrey Xu Yu, and Wenjie Zhang. 2015. Index-based Optimal Algorithms for Computing Steiner Components with Maximum Connectivity. In *Proc. of SIGMOD'15*.
[8] Lijun Chang and Lu Qin. 2018. *Cohesive Subgraph Computation over Large Sparse Graphs*. Springer Series in the Data Sciences.
[9] Lijun Chang, Jeffrey Xu Yu, Lu Qin, Xuemin Lin, Chengfei Liu, and Weifa Liang. 2013. Efficiently computing k-edge connected components via graph decomposition. In *Proc. of SIGMOD'13*. 205–216.
[10] Moses Charikar. 2000. Greedy approximation algorithms for finding dense components in a graph. In *Proc. of APPROX'00*. 84–95.
[11] Dehua Cheng, Natali Ruchansky, and Yan Liu. 2018. Matrix completability analysis via graph k-connectivity. In *Proc. of AISTATS'18*. 395–403.
[12] James Cheng, Yiping Ke, Shumo Chu, and M. Tamer Özsu. 2011. Efficient core decomposition in massive networks. In *Proc. of ICDE'11*. 51–62.
[13] Jonathan Cohen. 2008. Trusses: Cohesive subgraphs for social network analysis. *National Security Agency Technical Report* (2008), 16.
[14] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. 2001. *Introduction to Algorithms*. McGraw-Hill Higher Education.
[15] Maximilien Danisch, T.-H. Hubert Chan, and Mauro Sozio. 2017. Large Scale Density-friendly Graph Decomposition via Convex Programming. In *Proc. of WWW'17*. 233–242.
[16] Wai Shing Fung, Ramesh Hariharan, Nicholas J. A. Harvey, and Debmalya Panigrahi. 2011. A general framework for graph sparsification. In *Proc. of STOC'11*. 71–80.
[17] Alan Gibbons. 1985. *Algorithmic Graph Theory*. Cambridge University Press.
[18] A. V. Goldberg. 1984. *Finding a Maximum Density Subgraph*. Technical Report. Berkeley, CA, USA.
[19] R. E. Gomory and T. C. Hu. 1961. Multi-Terminal Network Flows. *J. Soc. Indust. Appl. Math.* 9, 4 (1961). http://dx.doi.org/10.2307/2098881
[20] Jiafeng Hu, Xiaowei Wu, Reynold Cheng, Siqiang Luo, and Yixiang Fang. 2017. On Minimal Steiner Maximum-Connected Subgraph Queries. *IEEE Trans. Knowl. Data Eng.* 29, 11 (2017), 2455–2469.
[21] Jonathan A. Kelner, Yin Tat Lee, Lorenzo Orecchia, and Aaron Sidford. 2013. An Almost-Linear-Time Algorithm for Approximate Max Flow in Undirected Graphs, and its Multicommodity Generalizations. In *Proc. of SODA'13*.
[22] David W. Matula and Leland L. Beck. 1983. Smallest-Last Ordering and clustering and Graph Coloring Algorithms. *J. ACM* 30, 3 (1983), 417–427.
[23] An Nguyen and Seok-Hee Hong. 2017. k-core based multi-level graph visualization for scale-free networks. In *Proc. of PacificVis'17*. 21–25.
[24] James B. Orlin. 2013. Max flows in O(nm) time, or better. In *Proc. of STOC'13*. 765–774.
[25] Apostolos N. Papadopoulos, Apostolos Lyritsis, and Yannis Manolopoulos. 2008. SkyGraph: an algorithm for important subgraph discovery in relational graphs. *Data Min. Knowl. Discov.* 17, 1 (Aug. 2008), 20. https://doi.org/10.1007/s10618-008-0109-y
[26] Kazumi Saito and Takeshi Yamada. 2006. Extracting Communities from Complex Networks by the k-dense Method. In *Proc. of ICDMw'06*. 300–304.
[27] Ahmet Erdem Sariyüce and Ali Pinar. 2016. Fast Hierarchy Construction for Dense Subgraphs. *PVLDB* 10, 3 (2016), 97–108.
[28] Stephen B. Seidman. 1983. Network structure and minimum degree. *Social Networks* 5, 3 (1983), 269 – 287.
[29] Jonah Sherman. 2013. Nearly Maximum Flows in Nearly Linear Time. In *Proc. of FOCS'13*.
[30] Manuel Sorge and et al. 2013. The graph parameter hierarchy.
[31] Bintao Sun, Maximilien Danisch, T.-H. Hubert Chan, and Mauro Sozio. 2020. KClist++: A Simple Algorithm for Finding k-Clique Densest Subgraphs in Large Graphs. *Proc. VLDB Endow.* 13, 10 (2020), 1628–1640.
[32] Dong Wen, Lu Qin, Ying Zhang, Lijun Chang, and Ling Chen. 2019. Enumerating k-Vertex Connected Components in Large Graphs. In *Proc. of ICDE'19*. 52–63.
[33] Douglas R. White and Frank Harary. 2001. The cohesiveness of blocks in social networks: Node connectivity and conditional density. *Sociological Methodology* 31 (2001).
[34] Xifeng Yan, X. Jasmine Zhou, and Jiawei Han. 2005. Mining closed relational graphs with connectivity constraints. In *Proc. of KDD'05* (Chicago, Illinois, USA). 10. https://doi.org/10.1145/1081870.1081908
[35] Long Yuan, Lu Qin, Xuemin Lin, Lijun Chang, and Wenjie Zhang. 2017. I/O efficient ECC graph decomposition via graph reduction. *VLDB J.* 26, 2 (2017). https://doi.org/10.1007/s00778-016-0451-4
[36] Rui Zhou, Chengfei Liu, Jeffrey Xu Yu, Weifa Liang, Baichen Chen, and Jianxin Li. 2012. Finding Maximal k-Edge-Connected Subgraphs from a Large Graph. In *Proc. of EDBT'12*.

A MORE DETAILS OF ECO-TD AND ECO-BU

Description of ECo-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).

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 have 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 ECo-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.

Running Examples. The following two running examples are for ECo-TD and ECo-BU, respectively.

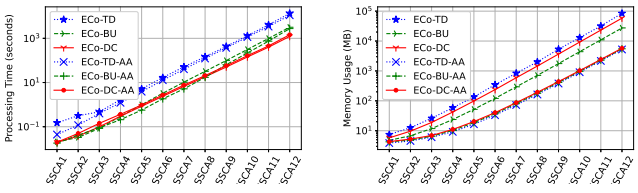
Example 1.1: Consider the graph in Figure 2. $\phi_2(G) = \{G\}$; thus, the weight of the root node r of $\mathcal{T}$ is 2. $\phi_3(G) = \{g_1 \oplus g_2, g_3\}$; thus, r has two children corresponding to $g_1 \oplus g_2$ and g_3, respectively. $\phi_4(g_3) = \emptyset$; thus, all vertices of g_3 are 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 Figure 3. □

Example 1.2: Consider the graph in Figure 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. 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 Figure 3. □

B ADDITIONAL EXPERIMENTAL RESULTS

We also evaluate the algorithms on SSCA graphs that are generated by the graph generator GTGraph [10]. An SSCA graph contains a collection of randomly sized cliques and also random inter-clique edges. We generate twelve 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

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



(a) Processing time (SSCA graphs) (b) Memory usage (SSCA graphs)

Figure 16: Results of ECo-decomposition on SSCA graphs

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.

The processing time and memory usage of the six ECo-decomposition algorithms on SSCA graphs are respectively shown in Figure 16(a) and Figure 16(b), 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.

C MISSING PROOFS

Proof of Lemma 3.1. 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. □

Proof of Property 1. 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. □

Proof of Property 2. This property directly follows from Property 1 and the definition of steiner connectivity. □

Proof of Property 3. This property directly follows from the properties 1 and 2, and the hierarchical structure of k-ECCs for different k values. □

Proof of Lemma 4.1. 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 $L < M \le H$ (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. □

Proof of Theorem 4.1. 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 4.1, 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 as shown in Figure 6. 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. □

Proof of Theorem 4.2. 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 $O(\mathsf{T}_{\mathtt{KECC}}(G))$, by assuming that $\mathsf{T}_{\mathtt{KECC}}(G)$ is linear or super-liner to the number of edges of G. Consequently, the total time complexity of Algorithm 3 is $O(h \times \mathsf{T}_{\mathtt{KECC}}(G))$ as the recursion tree has h levels. □

Proof of Theorem 4.3. 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 $O(\log \delta(G))$, as the interval for the root node is $[1, \delta(G)]$. Consequently, the time complexity of Algorithm 3 is $O\big((\log \delta(G)) \times \mathsf{T}_{\mathtt{KECC}}(G)\big)$ by following Theorem 4.2. □