I/O Efficient ECC Graph Decomposition via Graph Reduction

Long Yuan[§], Lu Qin[‡], Xuemin Lin[§], Lijun Chang[§], and Wenjie Zhang[§]
[§] The University of New South Wales, Australia
[‡]Centre for Quantum Computation & Intelligent Systems, University of Technology, Sydney, Australia
[§]{longyuan,lxue,ljchang,zhangw}@cse.unsw.edu.au; [‡]lu.qin@uts.edu.au

ABSTRACT

The problem of computing k-edge connected components (k-ECCs) of a graph G for a specific k is a fundamental graph problem and has been investigated recently. In this paper, we study the problem of ECC decomposition, which computes the k-ECCs of a graph G for all k values. ECC decomposition can be widely applied in a variety of applications such as graph-topology analysis, community detection, Steiner component search, and graph visualization. A straightforward solution for ECC decomposition is to apply the existing k-ECC computation algorithm to compute the k-ECCs for all k values. However, this solution is not applicable to large graphs for two challenging reasons. First, all existing k-ECC computation algorithms are highly memory intensive due to the complex data structures used in the algorithms. Second, the number of possible k values can be very large, resulting in a high computational cost when each k value is independently considered. In this paper, we address the above challenges, and study I/O efficient ECC decomposition via graph reduction. We introduce two elegant graph reduction operators which aim to reduce the size of the graph loaded in memory while preserving the connectivity information of a certain set of edges to be computed for a specific k. We also propose three novel I/O efficient algorithms, Bottom-Up, Top-Down, and Hybrid, that explore the k values in different orders to reduce the redundant computations between different k values. We analyze the I/O and memory costs for all proposed algorithms. In our experiments, we evaluate our algorithms using seven real large datasets with various graph properties, one of which contains 1.95 billion edges. The experimental results show that our proposed algorithms are scalable and efficient.

1. INTRODUCTION

Graphs have been widely used to represent the relationships of entities in real-world applications such as social networks, web search, collaborations networks, and biology. With the proliferation of graph applications, research efforts have been devoted to many fundamental problems in managing and analyzing graph data. Among them, the problem of computing all *k-Edge Connected Components* (k-ECCs) of a graph for a given k has been recently studied in [26, 32, 5, 9]. Here, a k-ECC of a graph G is a

Proceedings of the VLDB Endowment, Vol. 9, No. 7
Copyright 2016 VLDB Endowment 2150-8097/16/03.

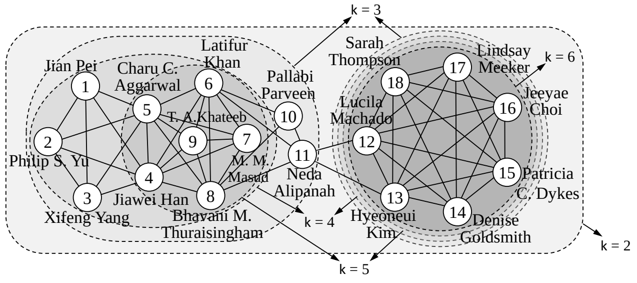


Figure 1: Part of the Coauthor Network

maximal subgraph g of G such that g is *k-edge connected* (i.e., g is connected after the removal of any $(k-1)$ edges from g).

Computing k-ECCs has many applications. For example, k-ECCs are used in social network analysis to discover cohesive blocks (communities) in a social network (e.g., Facebook) [25]. Computing the components with high connectivity is used to identify closely related entities in social behavior mining [4]. In computational biology, a highly connected subgraph is a functional cluster of genes in gene microarray study [10]. Computing k-ECCs can be used to identify groups of researchers with similar research interests in a collaboration network (e.g., DBLP). Moreover, k-ECCs computation also plays a role as a building block in many other applications such as the robust detection of communication networks and graph visualization [5, 9, 24, 27].

ECC **Decomposition**. In this paper, we study the ECC decomposition problem, which is to compute the k-ECCs of a graph for all possible k values. We give an example below:

Example 1.1: Fig. 1 shows a graph G, which is part of the collaboration network in the Coauthor dataset (http://arnetminer.org/). We compute the k-ECCs of G for all $2 \leq k \leq 6$. Here, G itself is a 2-ECC since after removing any edge from G, G is still connected. G has two 3-ECCs, which are the subgraphs induced by $\{v_1, v_2, \dots, v_{11}\}$ and $\{v_{12}, v_{13}, \dots, v_{18}\}$ respectively. The subgraph induced by $\{v_{12}, v_{13}, \dots, v_{18}\}$ is also a 4, 5, and 6-ECC of G. The subgraph induced by $\{v_1, v_2, \dots, v_9\}$ is a 4-ECC, and the subgraph induced by $\{v_4, v_5, \dots, v_9\}$ is a 5-ECC. As shown in Fig. 1, when k increases, the cohesiveness of the k-ECCs increases, whereas the size of the k-ECCs decreases. □

Using ECC decomposition, we can analyze the k-ECCs of a graph for all the k values rather than a specific k to better understand the network structure in each of the above-mentioned applications. Furthermore, ECC decomposition can also be used in many new application scenarios. For example:

- *Hierarchy Study in Networks.* The k-ECCs of a graph for all k values form a hierarchical structure. Understanding this hierarchical structure facilitates graph-topology analysis. In the literature, approximation techniques have been used to compute the graph connectivity hierarchy in [6, 7], and it is clear that ECC decomposition can solve the problem accurately.

- *Adaptive Community Detection*. Computing k-ECCs with high connectivity can be used to detect cohesive blocks (communities) in a social network [25]. However, it is not easy for a user to choose the best k. ECC decomposition can help the user to choose the best k adaptively according to the user's requirement.
- *Steiner Component Search*. In many applications, users may often want to find a subgraph with maximum connectivity that contains a given set of query nodes [8]. Such a subgraph is called a Steiner component. ECC decomposition can be used as a preprocessing step for the Steiner component search problem.
- *Multi-granularity Graph Visualization*. When applying k-ECCs in graph visualization [24, 27], users may want to visualize the graph in different granularities by zoom in and zoom out operations. ECC decomposition can be used directly to solve this multi-granularity graph visualization problem.

Challenges. Given a graph G, a straightforward solution for ECC decomposition is to independently compute the k-ECCs of G for all k values using a k-ECC computation algorithm [26, 32, 5, 9]. However, this solution presents the following two challenges:

Challenge 1: High Memory Consumption. All existing k-ECC computation algorithms assume that the graph G is retained in memory. In order to compute the k-ECCs of a graph G efficiently, they have to maintain complex data structures that have high memory cost. For example, on the *Orkut* dataset (a social network) with only 117.2 million edges used in our experiment, the state-of the art algorithm [9] consumes 15.4 GB memory for ECC decomposition. On the other hand, the size of many real-world graphs is huge. For example, the Facebook social network contains 1.32 billion nodes and 140 billion edges[1]; and a sub-domain of the web graph Clubweb12 contains 978.5 million nodes and 42.6 billion edges[2]. Therefore, applying the existing k-ECC computation algorithm on G directly is not scalable for handling large graphs because of the high memory consumption.

Challenge 2: High Computational Cost. In many real-world graphs, the maximum k value can be very large. For example, on the *sk-2005* dataset used in our experiment, the maximum k value reaches $4,510$. Applying the k-ECC computation algorithm for all k values independently will result in high computational cost, since large redundant computations will be produced due to the overlapping of k-ECCs for different k values.

Our Solution. In this paper, we focus on I/O efficient ECC decomposition. Targeting Challenge 1, we aim to reduce the memory used to compute the k-ECCs so that it can handle real-world graphs even when the memory is inadequate. Targeting Challenge 2, we aim to reduce the redundant k-ECC computations between different k values to improve the efficiency of the algorithm. To achieve this, we define an edge set $E_{\phi=k}(G)$ for each k value, which is the set of edges in the k-ECC of G, but not in the $(k+1)$-ECC of G. Due to the hierarchical structure of k-ECCs for all k values, the problem of ECC decomposition of G is equivalent to computing $E_{\phi=k}(G)$ for all k values. The benefits of computing $E_{\phi=k}(G)$ are twofold:

First (regarding Challenge 1), we observe that the size of $E_{\phi=k}(G)$ is usually much smaller than the size of G and is usually memory-resident. For example, in the *uk-2005* dataset with 936.36 million edges used in our experiment, the maximum size of $E_{\phi=k}(G)$ is only 15.69 million, which is 1.6% of the graph size. However, it is not easy to obtain $E_{\phi=k}(G)$ from G directly. Therefore, we define a k-edge connectivity preserved graph (k-PG), which is a graph G' such that $E_{\phi=k}(G) = E_{\phi=k}(G')$. Suppose that the k-PG is memory-resident and can be computed in an I/O efficient manner, we can now obtain $E_{\phi=k}(G)$ by computing $E_{\phi=k}(k\text{-PG})$ in memory.

Second (regarding Challenge 2), although the k-ECCs for different k values overlap, it is easy to see that the $E_{\phi=k}(G)$ for different k values are non-overlapping. Therefore, when computing $E_{\phi=k}(G)$ for all k values, the redundant computations can be largely reduced if the k-PG is carefully selected and computed.

To make our idea practically applicable, the following issues need to be addressed: (1) How can a good k-PG be obtained in an I/O efficient manner? and (2) How can the CPU and I/O costs be shared when computing the k-PGs for all k values?

Contributions. In this paper, we answer the above questions and make the following contributions.

(1) The first work for I/O efficient ECC *decomposition.* In this paper, we aim to solve the ECC decomposition problem on web-scale graphs by considering I/O issues when the memory size is inadequate. To the best of our knowledge, this is the first work to study the problem of I/O efficient ECC decomposition.

(2) Two elegant graph reduction operators to reduce memory usage. Our general idea to reduce the memory usage is graph reduction. We introduce two elegant graph reduction operators, RE and CE, for the removal and contraction of edges respectively. We discuss how to use these two graph reduction operators to minimize the size of the graph (k-PG) that preserves the connectivity information of the edges to be computed.

(3) Three novel I/O efficient algorithms by considering cost sharing. We derive three algorithms to compute the k-PGs for all k values, through which all k-ECCs can be computed. We discuss the potential cost sharing of k-PG computation when we explore k in different orders. Our Bottom-Up algorithm explores k in increasing order and eliminates edges with high connectivity when computing the k-PG. Our Top-Down algorithm explores k in decreasing order and eliminates edges with low connectivity when computing the k-PG. Our Hybrid algorithm takes advantage of both Bottom-Up and Top-Down and can minimize the size of the k-PG. In each algorithm, we also discuss how to compute the k-PG in an I/O efficient manner.

(4) Extensive performance studies on seven large real datasets. We conduct extensive performance studies using seven large real graphs with various graph properties. The experimental results demonstrate that our proposed algorithms can handle graphs with billions of edges using limited memory.

2. PRELIMINARIES

Consider an undirected graph $G = (V, E)$, where $V(G)$ represents the set of nodes and $E(G)$ represents the set of edges in G. We denote the number of nodes and the number of edges of G by n and m respectively. We define the size of G, denoted by $|G|$, as $|G| = m + n$. For each node $u \in V(G)$, we use $d(u, G)$ to denote degree of u. For simplicity, we use $d(u)$ to denote $d(u, G)$ if the context is self-evident. Given a set of nodes $V_n \subseteq V$, the node-induced subgraph by V_n, denoted by $G(V_n) = (V_n, E_n)$, is a subgraph of G such that $G(V_n) = (V_n, \{(u, v) \in E | u, v \in V_n\})$. Given a set of graphs $\mathbb{G} = \{G_1, G_2, \ldots, G_n\}$, $V(\mathbb{G}) = \bigcup_{i=1}^{n} V(G_i)$, $E(\mathbb{G}) = \bigcup_{i=1}^{n} E(G_i)$.

[1] http://newsroom.fb.com/company-info

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

Definition 2.1: (Edge-based Graph Connectivity) For a connected graph G, the edge-based graph connectivity of G, denoted by $\lambda(G)$, is the minimum number of edges whose removal makes G disconnected. □

Definition 2.2: (k-edge Connected) A connected graph G is k-edge connected iff the remaining graph is still connected after the removal of any $k-1$ edges from G. □

According to Definition 2.1 and Definition 2.2, a connected graph G is k-edge connected for any $1 \leq k \leq \lambda(G)$.

Definition 2.3: (k-edge Connected Component) Given a graph G, a subgraph G' of G is a k-edge connected component iff 1) G' is k-edge connected, and 2) any super-graph of G' in G is not k-edge connected. For simplicity, we use k-ECC as the abbreviation for the k-Edge Connected Component. □

In this paper, we use $\mathcal{C}_k(G)$ to denote the set of k-ECCs in graph G. For example: in Fig. 1, $\mathcal{C}_5(G)$ contains two 5-edge connected components induced by $\{v_4, v_5, \ldots, v_9\}$ and $\{v_{12}, v_{13}, \ldots, v_{18}\}$.

Problem Statement. In this paper, we study the problem of edge connected component (ECC) decomposition, which is defined as follows: Given a graph G, ECC decomposition computes the k-ECCs of G for all $2 \leq k \leq k_{max}$, where k_{max} is the maximum possible k value. Since the k-ECC computation operation is memory consuming, we aim to minimize the memory usage and focus on designing I/O efficient algorithms to compute the k-ECCs for all k values in the graph G.

When analyzing the I/O complexity of our algorithms, we use the standard I/O complexity notations in [2] as follows: M is the main memory size and B is the disk block size. The I/O complexity to scan N elements is $scan(N) = \Theta(\frac{N}{B})$, and the I/O complexity to sort N elements is $sort(N) = O(\frac{N}{B} \cdot \log_{\frac{M}{B}} \frac{N}{B})$.

The In-memory Algorithms. In the literature, there are several in-memory algorithms to compute k-ECCs for a specific k [26, 32, 5, 9]. In the following, we use Mem-Decom to denote the in-memory algorithm that computes k-ECCs for a specific k. The state-of-the-art in-memory algorithm [9] is based on a graph decomposition paradigm. For a given graph G and an integer k, a non k-edge connected subgraph of G is iteratively decomposed into several connected subgraphs by the removal of edges in all cuts of G with values less than k. The time complexity of the algorithm is $O(h \cdot l \cdot |E|)$ where h and l are usually bounded by small constants.

Based on Mem-Decom, a naive solution for solving the ECC decomposition problem is to use Mem-Decom to compute the corresponding k-ECCs on G directly for all possible k values. However, this solution has two drawbacks. First, due to the complex data structures used in Mem-Decom, this solution usually consumes a large amount of memory and is not scalable for large graphs. For example, on the *Orkut* dataset with only 117.2 million edges used in our experiment, this solution using the state-of-the-art algorithm [9] consumes 15.4 GB memory for ECC decomposition. Second, computing the k-ECCs for each k value individually is costly. Although some simple heuristics are used in [8] to compute all k-ECCs of a graph, the overlapping of k-ECCs for different k values, which is critical for reducing the overall computational cost, is not considered. Therefore, in this paper, we focus on I/O efficient issues to reduce the size of the memory used for ECC decomposition and we try to minimize redundant computation in ECC decomposition to reduce the CPU and I/O costs.

3. I/O EFFICIENT ECC DECOMPOSITION

In this section, we present the general idea of our algorithms. We first define a k-edge connectivity preserved graph k-PG and analyze the problem. Then, we give an overview of our algorithms. We summarize the notions used in this paper in Table 1.

3.1 k-edge Connectivity Preserved Graph

We define the edge connectivity number and connectivity bounded edge-set as follows:

Definition 3.1: (Edge Connectivity Number) Given a graph G and an edge e, the edge connectivity number of e, denoted by $\phi(e, G)$, is defined as $\phi(e, G) = \max\{k : e \in E(\mathcal{C}_k(G))\}$. We use k_{max} to denote the maximum edge connectivity number of edges in G, i.e. $k_{max} = \max_{e \in E(G)}\{\phi(e, G)\}$. □

Definition 3.2: (Connectivity Bounded Edge-Set) Given a graph G and a condition $f(\phi)$ on the edge connectivity number, the connectivity bounded edge-set, denoted by $E_{f(\phi)}(G)$, is the set of edges whose edge connectivity number $\phi(e, G)$ satisfies $f(\phi)$. □

For example, given a graph G and the condition $\phi = k$, $E_{\phi=k}(G)$ consists of edges whose edge connectivity number is k, i.e. $E_{\phi=k}(G) = \{e | e \in E(G), \phi(e, G) = k\}$. For simplicity, when the context is self-evident, we use $\phi(e)$ and $E_{f(\phi)}$ to denote $\phi(e, G)$ and $E_{f(\phi)}(G)$, respectively. With $\phi(e)$ for all $e \in E(G)$, the k-ECCs of G can be constructed based on:

Proposition 3.1: *For a given graph G, the k-edge connected component set $\mathcal{C}_k(G)$ consists of the subgraphs of G constructed by edges in $E_{\phi \geq k}(G)$.* □

Based on Proposition 3.1, we can deduce that if we can compute $\phi(e, G)$ for each $e \in E(G)$, we can construct all k-edge connected components easily by $E_{\phi \geq k}(G)$ for any $2 \leq k \leq k_{max}$. Since the sets $E_{\phi=k}(G)$ for different k values are non-overlapping, if we can compute $E_{\phi=k}(G)$ for every $2 \leq k \leq k_{max}$, then we can solve the ECC decomposition problem. Therefore, we provide an alternative problem definition as follows:

Definition 3.3: (Problem Definition*) Given a graph G, ECC decomposition computes $E_{\phi=k}(G)$ for any $2 \leq k \leq k_{max}$. □

Recall that the sets $E_{\phi=k}(G)$ for different k values are non-overlapping. Therefore, by computing $E_{\phi=k}(G)$ only, we have more possibilities for minimizing the redundant computations than computing the k-ECCs for all k values. Based on Definition 3.3, we define the k-edge connectivity preserved graph as follows:

Definition 3.4: (k-edge Connectivity Preserved Graph k-PG) Given a graph G and an integer k, a k-edge Connectivity Preserved Graph (k-PG) G' is a graph such that $E_{\phi=k}(G') = E_{\phi=k}(G)$. □

With Definition 3.4, to compute $E_{\phi=k}(G)$, we can construct a k-edge Connectivity Preserved Graph (k-PG) G' of G, and compute $E_{\phi=k}(G')$ using the in-memory algorithm. We aim to reduce the size of the k-PG in order to minimize memory usage.

3.2 Problem Analysis

To reduce the size of the k-PG, we define the following two types of graph reduction operators:

Definition 3.5: (Operator RE(G, E_r)) Given a graph G and a set of edges $E_r = (e_1, e_2, \ldots)$, RE(G, E_r) generates a new graph G_r by removing all the edges in E_r and all the nodes with degree 0 after removing the edges in E_r. □

Definition 3.6: (Operators CE(G, e) and CE(G, E_c)) Given a graph G and an edge $e = (u, v)$, CE(G, e) removes e, merges u and v into a new vertex v', and revises each edge e' incident to either u or v to be incident to v'. Given a graph G and a set of edges $E_c = (e_1, e_2, \ldots)$, CE(G, E_c) generates a new graph by applying CE(G, e) on all edges $e \in E_c$. □

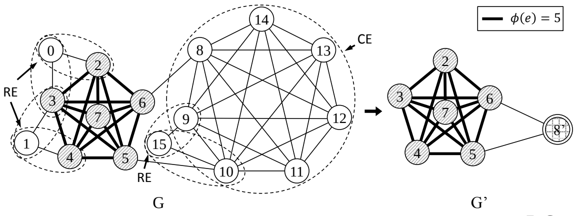


Figure 2: A 5-edge Connectivity Preserved Graph (5-PG)

Symbol	Description
$G = (V, E)$	graph with nodes V ane edges E
$d(u, G)$	The number of neighbors of u in G
$\mathcal{C}_k(G)$	the set of k-edge connected components of G
$\phi(e, G)$	the edge connectivity number of e in G
k_{max}	the maximum edge connectivity number of edges in G
$E_{f(\phi)}(G)$	the set of edges in G whose $\phi(e)$ satisfies $f(\phi)$
k-PG	k-edge connectivity preserved graph
U_k	the set of unprocessed connectivity numbers before a certain k
G_k	the input graph before processing a certain connectivity number k
$\underline{\phi}(e)$	lower bound of $\phi(e)$
$G^{k}_{\underline{c}ert}$	the union of $(k+1)$ edge-disjoint spanning forests of a graph
$\overline{\phi}(e)$	upper bound of $\phi(e)$
degree(e, G)	the edge degree number of e in G

Table 1: Notations

Note that after applying $\mathsf{CE}(G, E_c)$, parallel edges may be created. Using the graph reduction operators $\mathsf{RE}(G, E_r)$ and $\mathsf{CE}(G, E_c)$, we devise the following two propositions:

Proposition 3.2: *Given a graph G and a certain k,* $\mathsf{RE}(G, E_{\phi<k})$ *is a k*-PG *of G, i.e.,* $E_{\phi=k}(G) = E_{\phi=k}(\mathsf{RE}(G, E_{\phi<k}))$. □

Proposition 3.3: *Given a graph G and a certain k,* $\mathsf{CE}(G, E_{\phi>k})$ *is a k*-PG *of G, i.e.,* $E_{\phi=k}(G) = E_{\phi=k}(\mathsf{CE}(G, E_{\phi>k}))$. □

By combining Proposition 3.2 and Proposition 3.3, we have:

Corollary 3.1: *Given a graph G and a certain k,* $\mathsf{CE}(\mathsf{RE}(G, E_{\phi<k}), E_{\phi>k})$ *is a k*-PG *of G, i.e.,* $E_{\phi=k}(G) = E_{\phi=k}(\mathsf{CE}(\mathsf{RE}(G, E_{\phi<k}), E_{\phi>k}))$. □

Note that graph $\mathsf{CE}(\mathsf{RE}(G, E_{\phi<k}), E_{\phi>k})$ contains exactly the same set of edges in $E_{\phi=k}$. Therefore, $\mathsf{CE}(\mathsf{RE}(G, E_{\phi<k}), E_{\phi>k})$ is an optimal k-PG. However, computing this k-PG I/O efficiently is not easy. In this paper, instead of computing $E_{\phi<k}$ and $E_{\phi>k}$, we compute two sets $E'_{\phi<k} \subseteq E_{\phi<k}$ and $E'_{\phi>k} \subseteq E_{\phi>k}$. We can derive the following proposition easily.

Proposition 3.4: *Given a graph G and k, for any* $E'_{\phi<k} \subseteq E_{\phi<k}$ *and* $E'_{\phi>k} \subseteq E_{\phi>k}$, $\mathsf{CE}(\mathsf{RE}(G, E'_{\phi<k}), E'_{\phi>k})$ *is a k*-PG *of G, i.e.,* $E_{\phi=k}(G) = E_{\phi=k}(\mathsf{CE}(\mathsf{RE}(G, E'_{\phi<k}), E'_{\phi>k}))$. □

We try to maximize both $|E'_{\phi<k}|$ and $|E'_{\phi>k}|$ in an I/O efficient manner to minimize the size of $\mathsf{CE}(\mathsf{RE}(G, E'_{\phi<k}), E'_{\phi>k})$. We illustrate this idea using the following example:

Example 3.1: Consider the graph G shown in Fig. 2. Suppose, for instance, $k = 5$. The edges with edge-connectivity number 5 in G are the edges in the subgraph induced by nodes $\{v_2, v_3, \ldots, v_7\}$. After applying $\mathsf{CE}(\mathsf{RE}(G, E_r), E_c)$ where $E_r = \{(v_0, v_2), (v_0, v_3), (v_1, v_3), (v_1, v_4), (v_9, v_{15}), (v_{10}, v_{15})\}$ and E_c consists of the edges in the subgraph induced by $\{v_8, v_9, \ldots, v_{14}\}$, we can obtain the graph G', which is shown on the right side of Fig. 2. Since $E_r \subseteq E_{\phi<5}$ and $E_c \subseteq E_{\phi>5}$, according to Proposition 3.4, we have $E_{\phi=5}(G) = E_{\phi=5}(\mathsf{CE}(\mathsf{RE}(G, E_r), E_c))$, i.e., $\mathsf{CE}(\mathsf{RE}(G, E_r), E_c)$ is a 5-PG of G. □

3.3 Solution Overview

In this subsection, we give an overview of our solution. As shown in Section 3.2, we need to compute $E_{\phi=k}$ for each connectivity number $2 \leq k \leq k_{max}$, and try to maximize the computational cost sharing among different k values. To do this, we can reduce the input graph by removing unnecessary edges based on the already processed k values instead of using the original graph G as the input graph for each k value. To better describe our idea, we first provide the following definitions.

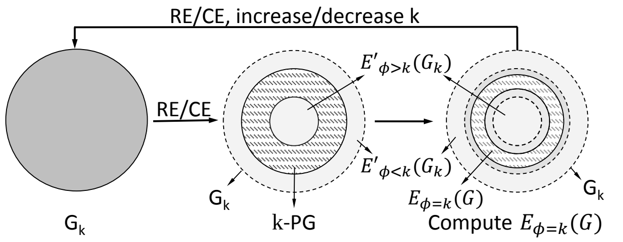


Figure 3: The Algorithm Framework

Algorithm 1 Bottom-Up(Graph G)

```
1: $k \leftarrow 1$; $G_k \leftarrow G$;
2: while $G_k \neq \emptyset$ do
3:     $k$-PG $\leftarrow \mathsf{CE}(G_k, E_{\underline{\phi}>k}(G_k))$;
4:     $E_{\phi=k} \leftarrow E(k\text{-PG}) \setminus E(\mathsf{Mem\text{-}Decom}(k\text{-PG}, k+1))$;
5:     $G_{k+1} \leftarrow \mathsf{RE}(G_k, E_{\phi=k})$;
6:     $k \leftarrow k+1$;
```

Definition 3.7: **(U_k, and G_k)** We use U_k to denote the set of unprocessed connectivity numbers before processing a certain connectivity number k, and use G_k to denote the input graph before processing a certain connectivity number k. □

Algorithm Framework. The framework of our approach is illustrated in Fig. 3. Given the input graph G_k for a certain k, we first apply the graph reduction operator RE/CE on G_k to compute the k-PG of G_k based on Proposition 3.4. Then we compute $E_{\phi=k}(G)$ on the k-PG using an in-memory algorithm. With $E_{\phi=k}(G)$, we refine the input graph by applying graph reduction operator RE/CE on G_k to generate the input graph for the next k value. The algorithm terminates when all k values have been processed.

To compute $E_{\phi=k}(G)$ for all the connectivity numbers $2 \leq k \leq k_{max}$ correctly using the framework shown in Fig. 3, the set of unprocessed connectivity numbers U_k and the input graph G_k for each k should satisfy the following two properties.

- (*Unseen-Connectivity Preservable*): For each connectivity number $i \in U_k$, $E_{\phi=i}(G) = E_{\phi=i}(G_k)$.
- (*Input-Graph Computable*): The input graph G_k can be computed by applying the reduction operators RE and CE on the input graph $G_{k'}$ for the previous iteration.

Following the framework, we propose three algorithms based on different orders of processing the connectivity numbers, namely, Bottom-Up, Top-Down, and Hybrid.

Algorithm Bottom-Up. The Bottom-Up algorithm computes all $E_{\phi=k}$ in increasing order of k. Therefore, we have $U_k = \{i | k \leq i \leq k_{max}\}$. We define the input graph G_k for a certain k to be the graph by removing all edges with $\phi < k$ using the RE operator, i.e., $G_k = \mathsf{RE}(G, E_{\phi<k})$. The *unseen-connectivity preservable* property and the *input-graph computable* property are satisfied by the following two propositions respectively:

Proposition 3.5: *Given a graph G and a connectivity number k, for any* $k \leq i \leq k_{max}$, $E_{\phi=i}(G) = E_{\phi=i}(\mathsf{RE}(G, E_{\phi<k}))$. □

Proposition 3.6: *Given a graph G and a connectivity number k,* $\mathsf{RE}(G, E_{\phi<k+1}) = \mathsf{RE}(\mathsf{RE}(G, E_{\phi<k}), E_{\phi=k})$. □

Intuitively, Proposition 3.5 follows the fact that the sets $E_{\phi=k}(G)$ for different k values are non-overlapping and removing

Algorithm 2 Top-Down(Graph G)

```
1: $k \leftarrow \overline{k}_{max}$; $G_k \leftarrow G$;
2: while $k > 1$ do
3:     $k$-PG $\leftarrow$ RE$(G_k, E_{\overline{\phi}<k}(G_k))$;
4:     $E_{\phi=k} \leftarrow E($Mem-Decom$(k$-PG$, k))$;
5:     $G_{k-1} \leftarrow$ CE$(G_k, E_{\phi=k})$;
6:     $k \leftarrow k-1$;
```

the edges with small edge connectivity number does not affect the values of edge connectivity number of the remaining edges. Proposition 3.6 is based on the property that $E(\mathcal{C}_k(G)) \subseteq E(\mathcal{C}_{k-1}(G))$ for any $2 \leq k \leq k_{max}$ and G_k can be computed according to G_{k-1} by RE operator.

To compute the k-PG for G_k, according to Proposition 3.4, we need to compute two sets $E'_{\phi<k} \subseteq E_{\phi<k}$ and $E'_{\phi>k} \subseteq E_{\phi>k}$. Since $G_k = \mathsf{RE}(G, E_{\phi<k})$, there is no edge with $\phi < k$ in G_k. Therefore, we only need to compute $E'_{\phi>k}$. However, the exact $\phi(e)$ values for edges e with $\phi(e) > k$ are hard to obtain. Therefore, we first compute a *lower bound* $\underline{\phi}(e)$ of $\phi(e)$ for each $e \in E(G_k)$. It is evident that $E_{\underline{\phi}>k} \subseteq E_{\phi>k}$. In this way, we can compute the k-PG by $\mathsf{CE}(G_k, E_{\underline{\phi}>k})$. We have:

Proposition 3.7: *For a certain connectivity number k, the k-PG for the* Bottom-Up *algorithm is* $\mathsf{CE}(\mathsf{RE}(G, E_{\phi<k}), E_{\underline{\phi}>k})$. □

The framework of Bottom-Up is shown in Algorithm 1. We start processing $k = 1$ and initially G_k is the original graph G (line 1). The algorithm iteratively increases k until $G_k = \emptyset$ (lines 2-6). In each iteration, for a certain k, we first compute the k-PG by Proposition 3.7 (line 3). Then, we can compute $E_{\phi=k}$ using $E(k$-PG$) \setminus E($Mem-Decom$(k$-PG, $k+1))$ (line 4), because according to Proposition 3.1, Mem-Decom$(k$-PG, $k+1)$ computes the set $E_{\phi\geq k+1}$, and the k-PG does not include edges in $E_{\phi<k}$. Here, $E_{\phi=k}$ is correctly computed because of Proposition 3.5. Lastly, we construct G_{k+1} for the next iteration based on Proposition 3.6.

<u>**Algorithm** Top-Down</u>. The Top-Down algorithm computes all $E_{\phi=k}$ in decreasing order of k. Therefore, we have $U_k = \{2 \leq i \leq k\}$. We define the input graph G_k for a certain k to be the graph by contracting all edges with $\phi > k$ using the CE operator, i.e., $G_k = \mathsf{CE}(G, E_{\phi>k})$. The *unseen-connectivity preservable* property and the *input-graph computable* property are satisfied by the following two propositions respectively:

Proposition 3.8: *Given a graph G and a connectivity number k, for any $2 \leq i \leq k$, $E_{\phi=i}(G) = E_{\phi=i}(\mathsf{CE}(G, E_{\phi>k}))$.* □

Proposition 3.9: *Given a graph G and a connectivity number k,* $\mathsf{CE}(G, E_{\phi>k-1}) = \mathsf{CE}(\mathsf{CE}(G, E_{\phi>k}), E_{\phi=k})$. □

Similar to Bottom-Up, to compute the k-PG for G_k in Top-Down, according to Proposition 3.4, we need to compute two sets $E'_{\phi<k} \subseteq E_{\phi<k}$ and $E'_{\phi>k} \subseteq E_{\phi>k}$. Since $G_k = \mathsf{CE}(G, E_{\phi>k})$, there is no edge with $\phi > k$ in G_k. Therefore, we only need to compute $E'_{\phi<k}$. However, the exact $\phi(e)$ values for edges e with $\phi(e) < k$ are hard to obtain. Therefore, we first compute an *upper bound* $\overline{\phi}(e)$ of $\phi(e)$ for each $e \in E(G_k)$. It is evident that $E_{\overline{\phi}<k} \subseteq E_{\phi<k}$. In this way, we can compute the k-PG by $\mathsf{RE}(G_k, E_{\overline{\phi}<k})$. We can derive the following proposition:

Proposition 3.10: *For a certain connectivity number k, the k-PG for the* Top-Down *algorithm is* $\mathsf{RE}(\mathsf{CE}(G, E_{\phi>k}), E_{\overline{\phi}<k})$. □

The framework of Top-Down is shown in Algorithm 2. Since k_{max} is unknown, we compute an upper bound $\overline{k}_{max}$ of k_{max}. We start processing $k = \overline{k}_{max}$ and initially G_k is the original graph G (line 1). The algorithm iteratively decreases k until $k \leq 1$ (lines 2-6). In each iteration, for a certain k, we first compute the k-PG by Proposition 3.10 (line 3). Then, we can compute $E_{\phi=k}$

Algorithm 3 Hybrid(Graph G)

```
1: $k \leftarrow \overline{k}_{max}$; $G_k \leftarrow G$;
2: while $k > 1$ do
3:     $G' \leftarrow$ RE$(G_k, E_{\overline{\phi}<k}(G_k))$;
4:     Compute the $k$-PG of $G'$ by Bottom-Up$(G')$;
5:     $E_{\phi=k} \leftarrow E(k$-PG$)$;
6:     $G_{k-1} \leftarrow$ CE$(G_k, E_{\phi=k})$;
7:     $k \leftarrow k-1$;
```

using $E($Mem-Decom$(k$-PG, $k))$ directly (line 4), because according to Proposition 3.1, Mem-Decom$(k$-PG, $k)$ computes the edge set $E_{\phi\geq k}$, and the k-PG does not include edges in $E_{\phi>k}$. Here, $E_{\phi=k}$ is correctly computed because of Proposition 3.8. Lastly, we construct G_{k-1} for the next iteration based on Proposition 3.9.

<u>**Algorithm** Hybrid</u>. Hybrid takes advantage of both Bottom-Up and Top-Down to further reduce the size of the k-PG. According to Proposition 3.10, the k-PG of the Top-Down algorithm contains the set of edges $E_{\overline{\phi}\geq k}(G_k)$ where $G_k = \mathsf{CE}(G, E_{\phi>k})$. In other words, the k-PG contains the edges e with $\overline{\phi}(e) \geq k$ and $\phi(e) \leq k$. Hybrid aims to further reduce the size of the k-PG by eliminating those edges with $\phi(e) < k$. The Bottom-Up algorithm can be naturally applied to handle this. The framework of Hybrid is shown in Algorithm 3. It generally follows the framework of Algorithm 2. However, after computing $G' = \mathsf{RE}(G_k, E_{\overline{\phi}<k}(G_k))$ in line 3, we do not use G' as the k-PG. Instead, we compute the k-PG of G' as the k-PG of G by invoking Bottom-Up(G) (line 4). Since by Proposition 3.7, Bottom-Up can remove all edges with $\phi < k$ when computing the k-PG, we can easily derive:

Proposition 3.11: *For a certain connectivity number k, the k-PG for the* Hybrid *algorithm is* $\mathsf{RE}(\mathsf{CE}(G, E_{\phi>k}), E_{\phi<k})$. □

In other words, Hybrid can compute the optimal k-PG. Furthermore, since the graph $\mathsf{CE}(\mathsf{RE}(G, E_{\phi<k}), E_{\phi>k})$ contains exactly the same set of edges in $E_{\phi=k}$, we can use $E(k$-PG$)$ as $E_{\phi=k}$ (line 5) without invoking Mem-Decom$(k$-PG $, k)$. The rationale for applying the Bottom-Up algorithm on G' is that G' can preserve the edges e with $\phi(e) = k$ according to Proposition 3.4. Note that by invoking Bottom-Up(G'), we also compute the set $E_{\phi=k'}(G')$ for any $2 \leq k' < k$. However, since G' does not satisfy the *unseen-connectivity preservable* property, this set on G' cannot be used as the result in the original graph G.

4. BOTTOM-UP DECOMPOSITION

In this section, we discuss Bottom-Up in detail. We first describe how to compute a tight $\underline{\phi}(e)$. Then we show how to implement Bottom-Up I/O efficiently. Lastly, we analyze the peak memory usage and I/O complexity of Bottom-Up.

4.1 $\underline{\phi}(e)$ Computation

As discussed in Section 3.3, the key issue to obtaining a good k-PG in Bottom-Up is to compute a tight $\underline{\phi}(e)$ for any edge e in the graph G. According to Definition 3.1, for an edge e in G, its edge connectivity number in G cannot be smaller than that in a subgraph of G, then a valid $\underline{\phi}(e)$ can be computed based on:

Proposition 4.1: *For any subgraph G_s of G and edge $e \in E(G_s)$, $\phi(e, G_s) \leq \phi(e, G)$.* □

By Proposition 4.1, we can select a subgraph G_s of G, and use $\phi(e, G_s)$ as $\underline{\phi}(e)$ for each $e \in E(G_s)$. However, arbitrarily selecting a subgraph G_s of G may result in a very loose $\underline{\phi}(e)$. Recall that in the Bottom-Up algorithm, the k-PG is computed using $\mathsf{CE}(G_k, E_{\underline{\phi}>k}(G_k))$. A loose $\underline{\phi}(e)$ may lead to a large k-PG when k becomes large. Nevertheless, in $\mathsf{CE}(G_k, E_{\underline{\phi}>k}(G_k))$, we only care about those edges e with $\underline{\phi}(e) > k$ in G_k when computing the

k-PG. Therefore, when k is small, although G_k is large, $\underline{\phi}(e)$ does not need to be very tight since $\underline{\phi}(e) > k$ can be easily satisfied by selecting a small subgraph of G_k. When k is large, G_k becomes small, and thus we can afford to select a subgraph of a large portion of G_k to compute a tight $\underline{\phi}(e)$.

Based on the above discussion, we can adaptively compute and update $\underline{\phi}(e)$ in G_k when k increases from 2 to k_{max}. We denote the subgraph used to compute $\underline{\phi}(e)$ in G_k as a *certificate graph* G^k_{cert}.

Certificate Graph G^k_{cert}. We construct the certificate graph G^k_{cert} from G_k as follows: Initially, $G^k_{cert} = (V(G_k), \emptyset)$. We construct G^k_{cert} using $k+1$ iterations. In each iteration, we first compute a spanning forest $\mathcal{F}$ of the graph with edges $E(G_k) \setminus E(G^k_{cert})$, and then update the edge set of G^k_{cert} to be $E(G^k_{cert}) \cup E(\mathcal{F})$. It is easy to derive the following proposition:

Proposition 4.2: $|E(G^k_{cert})| \leq (k+1) \times (|V(G_k)| - 1)$. □

The size of G^k_{cert} can be bounded because although $|V(G_k)|$ is large, we only need to load a small number of spanning forests of G_k to construct G^k_{cert} when k is small, and when k is large, $|V(G_k)|$ becomes small, thus we can load more spanning forests of G_k to construct G^k_{cert}.

Fig. 4 (a) shows a comparison of $|G|$, $|G_k|$ and $|G^k_{cert}|$ for Bottom-Up on the *uk-2005* dataset when we increase k from 2 to 100. $|G_k|$ decreases as k increases. For $|G^k_{cert}|$, we observe that, when k is small, $|G^k_{cert}|$ increases as k increases. After reaching the peak point with $k = 20$, $|G^k_{cert}|$ decreases as k increases. Notably, the peak size of G^k_{cert} is only around 20% of $|G|$, which is much smaller than $|G|$. Therefore, it is usually suitable to use G^k_{cert} to compute $\underline{\phi}(e)$ in G_k.

Computing $\underline{\phi}(e)$ for $e \in E(G^k_{cert})$. Since G^k_{cert} is the union of $(k+1)$ edge-disjoint spanning forests of a graph, we can derive the following proposition based on the theoretical result derived in [20] and Definition 2.2:

Proposition 4.3: *Given a graph G and k, for any $2 \leq i \leq k+1$, the graph G^k_{cert} of G is i-edge connected if G is i-edge connected.* □

Recall that the graph G_k of Bottom-Up is defined as $G_k = \mathsf{RE}(G, E_{\phi<k})$. Therefore, for each edge $e \in E(G_k)$, we have $\phi(e, G_k) \geq k$, i.e., G_k is k-edge connected. Since G^k_{cert} is constructed based on G_k, G^k_{cert} is also k-edge connected according to Proposition 4.3. Therefore, we have:

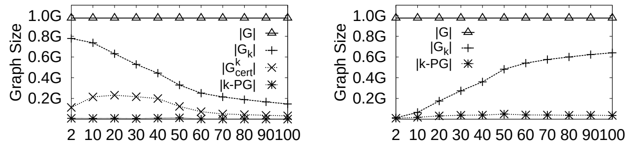
Corollary 4.1: *For each edge $e \in E(G^k_{cert})$, $\phi(e, G^k_{cert}) \geq k$.* □

Since $\underline{\phi}(e)$ varies in different G_k, we denote $\underline{\phi}(e)$ for G_k as $\underline{\phi}_k(e)$. With G^k_{cert}, for each $e \in E(G^k_{cert})$, $\underline{\phi}_k(e)$ can be simply computed as $\phi(e, G^k_{cert})$. However, to compute $\phi(e, G^k_{cert})$, we need to compute the k'-ECC for all $2 \leq k' \leq k_{max}$ in G^k_{cert}, which is costly. Recall that the aim of computing $\underline{\phi}_k(e)$ is to obtain the set $E_{\underline{\phi}>k}(G_k)$. Therefore, for an edge $e \in E(G^k_{cert})$, as long as we guarantee $\underline{\phi}_k(e) > k$, we do not need to compute the exact $\underline{\phi}_k(e)$. In other words, for each $e \in E(G^k_{cert})$, if we guarantee $\phi(e, G^k_{cert}) > k$, we can simply set $\underline{\phi}_k(e)$ as $k+1$ without computing $\phi(e, G^k_{cert})$. Based on this, we can define $\underline{\phi}_k(e)$ for each $e \in E(G^k_{cert})$ as follows:

$$\underline{\phi}_k(e) = \min\{\phi(e, G^k_{cert}), k+1\} \tag{1}$$

Based on Eq. 1 and Corollary 4.1, we have:

Corollary 4.2: *For each $e \in E(G^k_{cert})$, $k \leq \underline{\phi}_k(e) \leq k+1$.* □



(a) Vary k (Bottom-Up) (b) Vary k (Top-Down)

Figure 4: Size of Different Graphs on the *uk-2005* Dataset

According to the above discussion, for each edge $e \in E(G^k_{cert})$, we only need to compute the $(k+1)$-ECC of G^k_{cert} in memory to compute $\underline{\phi}_k(e)$. If e belongs to the $(k+1)$-ECC of G^k_{cert}, we can set $\underline{\phi}_k(e)$ to be $k+1$; otherwise, we set $\underline{\phi}_k(e)$ to be k. Note that our objective is to maximize the number of edges with $\underline{\phi}_k(e) = k+1$. By Corollary 4.2, for each $e \in E(G^k_{cert})$, $\underline{\phi}_k(e)$ is tight in the sense that $\underline{\phi}_k(e)$ only differs from $(k+1)$ by at most 1.

Computing $\underline{\phi}(e)$ for $e \notin E(G^k_{cert})$. Note that there are also edges in $E(G_k)$ that do not belong to $E(G^k_{cert})$. For each such edge $e = (u, v)$, if u and v belong to the same $(k+1)$-ECC of G^k_{cert}, u and v also belong to the same $(k+1)$-ECC of G_k, thus we can set $\underline{\phi}_k(e)$ to be $(k+1)$; otherwise, $\underline{\phi}_k(e)$ is set to be k since G_k itself is a k-ECC. The rationale for this is that, by combining $(k+1)$ edge-disjoint spanning forests of G_k, most parts of the $(k+1)$-ECCs of G_k are also preserved in G^k_{cert}. For example, on the *uk-2005* dataset with 39.45 million nodes and 936.36 million edges used in our experiment, 96.2% of nodes in the $(k+1)$-ECCs of G_k are preserved in G^k_{cert} on average. Based on this, $\underline{\phi}_k(e)$ can still be effectively computed for each edge $e \in E(G_k) \setminus E(G^k_{cert})$.

The General Case. Given G^k_{cert}, we can derive a general method to compute $\underline{\phi}_k(e)$ for each $e \in E(G_k)$ based on:

Corollary 4.3: *For each $e = (u, v)$ in $E(G_k)$, if u and v belong to the same $(k+1)$-ECC of G^k_{cert}, $\underline{\phi}_k(e) = k+1$; otherwise $\underline{\phi}_k(e) = k$.* □

Fig. 4 (a) shows the size of the k-PG constructed by computing $\underline{\phi}_k(e)$ using the above method in the *uk-2005* dataset when varying k from 2 to 100. For all k values, the size of the k-PG is much smaller than $|G|$ and even smaller than $|G^k_{cert}|$, which indicates that the $\underline{\phi}_k(e)$ values computed in this way are effective.

4.2 The Bottom-Up Decomposition Algorithm

In this subsection, we discuss how to implement Bottom-Up I/O efficiently. For simplicity, we assume that graph G_k ($2 \leq k \leq k_{max}$) is connected. Otherwise, we can handle each connected component of G_k individually.

The Bottom-Up algorithm is shown in Algorithm 4, which follows the framework of Algorithm 1 and processes k in its increasing order. We use M_{peak} to denote the peak memory usage of the algorithm. When G_k can be processed in M_{peak} memory ($|G_k| \times \alpha \leq M_{peak}$), we can just apply the in-memory algorithm following Algorithm 1 to compute $E_{\phi=k'}$ for all $k' > k$ (lines 3-5). Here, α is determined by the in-memory algorithm Mem-Decom used to compute the k-ECCs of a graph. If G_k cannot be processed in M_{peak} memory, we first compute G^k_{cert} by invoking procedure DisjointForest (line 6), and compute the k-PG using the CE operator by invoking procedure CE-Disk (lines 7-8). Then, we load the k-PG in memory, and after computing $E_{\phi=k}$ on the k-PG in memory (line 9), we compute G_{k+1} using the RE operator by invoking procedure RE-Disk. Below, we introduce the procedures DisjointForest, CE-Disk, and RE-Disk in detail.

Algorithm 4 Bottom-Up(Graph G)

```
1:  $G_k \leftarrow G$; $k \leftarrow 1$;
2:  while $G_k \neq \emptyset$ do
3:      if $|G_k| \times \alpha \leq M_{peak}$ then
4:          compute $E_{\phi=k'}(G_{k'})$ for $k' \geq k$ in memory following Algorithm 1;
5:          break;
6:      $G^k_{cert} \leftarrow$ DisjointForest($G_k$);
7:      $G' \leftarrow$ Mem-Decom($G^k_{cert}, k+1$);
8:      $k$-PG $\leftarrow$ CE-Disk($G_k, G'$);
9:      $E_{\phi=k} \leftarrow E(k\text{-PG}) \setminus E(\text{Mem-Decom}(k\text{-PG}, k+1))$;
10:     $G_{k+1} \leftarrow$ RE-Disk($G_k, E_{\phi=k}$);
11:     $k \leftarrow k+1$;
12: procedure DisjointForest(Disk Graph $G_k$)
13: $G^k_{cert} \leftarrow \emptyset$;
14: for $i = 0$ to $k$ do
15:     $\mathcal{F} \leftarrow \emptyset$;
16:     for all edge $(u,v) \in E(G_k)$ by sequential scanning $G_k$ on disk do
17:         if $(u,v) \notin G^k_{cert}$ and $u, v$ are not connected in $\mathcal{F}$ then
18:             $\mathcal{F} \leftarrow \mathcal{F} \cup (u,v)$;
19:     $G^k_{cert} \leftarrow G^k_{cert} \cup \mathcal{F}$;
20: return $G^k_{cert}$;
21: procedure CE-Disk(Disk Graph $G_k$, Graph $G'$)
22: $G_c \leftarrow \emptyset$ on disk;
23: create a node w.r.t. each connected component of $G'$ in memory;
24: for all edge $(u,v) \in E(G_k)$ by sequential scanning $G_k$ on disk do
25:     if $u \in V(G')$ then
26:         $w_u \leftarrow$ the node w.r.t. the connected component in $G'$ that contains $u$;
27:     else $w_u \leftarrow u$;
28:     if $v \in V(G')$ then
29:         $w_v \leftarrow$ the node w.r.t. the connected component in $G'$ that contains $v$;
30:     else $w_v \leftarrow v$;
31:     if $w_u \neq w_v$ then
32:         add edge $(w_u, w_v)$ in $G_c$ on disk;
33: return $G_c$;
34: procedure RE-Disk(Disk Graph $G_k$, Edge Set $E$)
35: $G_r \leftarrow \emptyset$ on disk;
36: for all edge $(u,v) \in E(G_k)$ by sequential scanning $G_k$ on disk do
37:     if $e \notin E$ then
38:         add edge $(u,v)$ in $G_r$ on disk;
39: return $G_r$;
```

Procedure DisjointForest. The DisjointForest procedure is used to compute G^k_{cert} of G_k (stored on disk). It initializes G^k_{cert} (line 13) and computes G^k_{cert} by scanning all edges in G_k sequentially on disk for $k+1$ times. In each scan (lines 15-19), a spanning forest is computed (lines 16-18) and added to G^k_{cert} (line 19). To compute a spanning forest of $E(G_k) \setminus E(G^k_{cert})$, we do not compute $E(G_k) \setminus E(G^k_{cert})$ explicitly as G_k needs to be scanned once more. Instead, for each edge $(u,v) \in E(G_k)$, we only need to check whether $(u,v) \in E(G^k_{cert})$ in memory. If $(u,v) \notin E(G^k_{cert})$, we further check whether u and v are connected in the current spanning forest $\mathcal{F}$ (line 17) using the union-find data structure in memory. If not, we add (u,v) to the spanning forest $\mathcal{F}$. After computing $\mathcal{F}$, we add it to G^k_{cert} (line 19). The procedure terminates and returns G^k_{cert} after $k+1$ disjoint spanning forests are added to G^k_{cert}.

Procedure CE-Disk. The procedure CE-Disk is used to compute the k-PG on G_k (stored on disk) by CE($G_k, E_{\underline{\phi}>k}(G_k)$). According to Corollary 4.3, to obtain $E_{\underline{\phi}>k}(G_k)$, we need to compute the $(k+1)$-ECC G' of G^k_{cert} (line 7). With G', we invoke CE-Disk(G_k, G') to compute the k-PG (line 8). In CE-Disk(G_k, G') (lines 21-33), based on Corollary 4.3, to contract edges with $\underline{\phi} > k$, we only need to compute the connected components of G' and contract the nodes in each connected component into one node in G_k to obtain CE($G_k, E_{\underline{\phi}>k}(G_k)$). To do so, we first create a node w.r.t. each connected component of G' in memory (line 23). Then we scan all edges $(u,v) \in E(G_k)$ sequentially on disk. If u (or v) is contracted to a new node, we revise the edge (u,v) by replacing u (or v) to be the corresponding contracted node (lines 25-30). We denote the revised edge as (w_u, w_v) and add it into the result graph G_c on disk if it is not a self-edge (i.e., $w_u \neq w_v$) (lines 31-32). Here, by revising (u,v) in G_k to be (w_u, w_v) in G_c, we still consider (u,v) and (w_u, w_v) as the same edge when they are compared. This can be implemented easily using node mapping. Lastly, after scanning all edges in G_k once, we can return G_c on disk as CE($G_k, E_{\underline{\phi}>k}(G_k)$) (line 33).

Proecedure RE-Disk. The procedure RE-Disk(G_k, E) is used to compute G_{k+1} (stored on disk) by operator RE(G_k, E) with $E = E_{\phi=k}$ on graph G_k (stored on disk). The procedure scans all edges of G_k sequentially on disk (line 36). For each edge (u,v), if $(u,v) \notin E$, (u,v) belongs to RE(G_k, E), and thus we add (u,v) to the result graph on disk (line 38). After scanning all edges in G_k once, we return the result graph on disk (line 39).

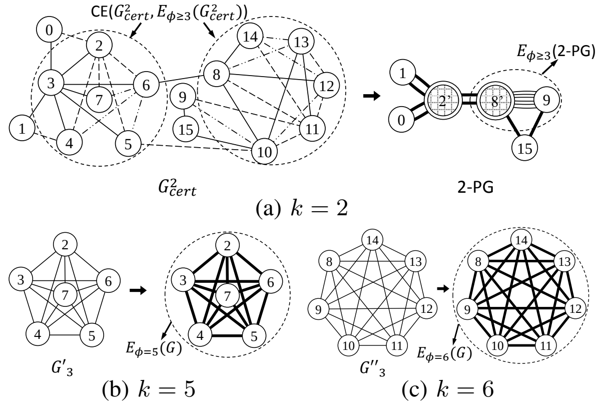


Figure 5: Bottom-Up Example

Example 4.1: Fig. 5 illustrates a running example of Bottom-Up. Consider the graph G in Fig. 2 as the input graph. For $k=2$, the input graph G_2 is G itself. We obtain G^2_{cert} by computing 3 edge-disjoint spanning forests which are illustrated with different types of lines in Fig. 5(a). Then we compute G' based on G^2_{cert}, which has two connected components and is highlighted with dotted circles in Fig. 5 (a). After contracting the connected components in G' on G_2, we obtain 2-PG. When we have obtained 2-PG, we compute $E_{\underline{\phi}\geq 3}$(2-PG) by Mem-Decom($G^2_{cert}, 3$). Then $E_{\phi=2}$(2-PG) is computed by E(2-PG) $\setminus E_{\underline{\phi}\geq 3}$(2-PG), which is $E_{\phi=2} = \{(v_0,v_2),(v_0,v_3),(v_1,v_3),(v_1,v_4),(v_6,v_8),(v_5,v_{10}),(v_9,v_{15}),(v_{10},v_{15})\}$. Then we remove $E_{\phi=2}$ from G_2 and move to $k=3$. Note that after removing $E_{\phi=2}$, the graph is divided into 2 subgraphs, namely the subgraphs induced by $\{v_2,v_3,\ldots,v_7\}$ (G'_3) and $\{v_8,v_9,\ldots,v_{14}\}$ (G''_3), respectively. Now, we can handle G'_3 and G''_3 individually. For $k=3,4$, the cases are trivial and $E_{\phi=3}=\emptyset$ and $E_{\phi=4}=\emptyset$. For $k=5,6$, $E_{\phi=5}$ consists of the edges in G'_3 and $E_{\phi=6}$ consists of edges in G''_3, which are shown in Fig. 5 (b) and Fig. 5 (c), respectively. □

Complexity Analysis. Below, we show the peak memory usage and I/O complexity of our Bottom-Up algorithm (Algorithm 4):

Theorem 4.1: *Given a graph G, let $M^{bu}_{cert}(G)$ be the maximum size of G^k_{cert}, $M^{bu}_{kpg}(G)$ be the maximum size of k-PG, and $M^{bu}(G)$ be the peak memory used in Bottom-Up (Algorithm 4), we have:*

(1) $M^{bu}_{cert}(G) = O(\max_{1\leq k\leq k_{max}}\{k\cdot|V(E_{\phi\geq k}(G))|\})$;

(2) $M^{bu}_{kpg}(G) = O(\max_{1\leq k\leq k_{max}}\{|E_{\phi\geq k,\underline{\phi}\leq k}(G)|\})$;

(3) $M^{bu}(G) = O(\max\{M^{bu}_{cert}(G), M^{bu}_{kpg}(G)\})$. □

Here, $|V(E_{\phi\geq k}(G))|$ is the number of nodes in the graph consisting of edges in $E_{\phi\geq k}(G)$. According to our discussion in Section 4.1, both $M^{bu}_{kpg}(G)$ and $M^{bu}_{cert}(G)$ are usually much smaller than $|G|$. Therefore, $M^{bu}(G)$ is usually much smaller than the memory consumed by the in-memory algorithm.

Theorem 4.2: *Given a graph G, let $I^{bu}(G)$ be the number of I/Os used in* Bottom-Up *(Algorithm 4), we have:*

$$I^{bu}(G) = O(\sum_{k=1}^{k_{max}} k \cdot scan(|E_{\phi \geq k}(G)|)). \qquad \square$$

__Discussion.__ Bottom-Up (Algorithm 4) exhibits the worst case behaviour when the input graph is a clique. In this case, when $k < k_{max}$, $E_{\phi=k} = \emptyset$ in line 9 and we cannot remove any edges in line 10. Then G_k is always the same as G when $k < k_{max}$. In this case, $M^{bu}(G) = O(|E(G)|)$ and $I^{bu}(G) = O(k_{max}^2 \cdot scan(|E(G)|)$.

5. TOP-DOWN DECOMPOSITION

In this section, we discuss Top-Down in detail. We first introduce how to compute a tight $\overline{\phi}(e)$. Then we show how to implement Top-Down I/O efficiently. Lastly, we analyze the peak memory usage and I/O complexity of Top-Down.

5.1 $\overline{\phi}(e)$ Computation

From the analysis of Section 3.3, we need to compute an upper bound $\overline{\phi}(e)$ for each $e \in E(G)$ to compute a good k-PG. In addition, $\overline{\phi}(e)$ should be computed I/O efficiently without introducing much extra I/O or memory cost. To achieve this, we first define the edge degree number as follows:

Definition 5.1: (Edge Degree Number degree(e, G)) For a given graph G and an edge $e = (u, v)$, the edge degree number of e, denoted by degree(e, G), is the minimum degree of u and v in G, i.e., degree$((u, v), G) = \min\{d(u, G), d(v, G)\}$. $\square$

We also use degree(e) to represent degree(e, G) when it is self-evident. Based on Definition 5.1, the following proposition holds:

Proposition 5.1: *Given a graph G and an edge $e \in E(G)$, we have* degree$(e, G) \geq \phi(e, G)$. $\square$

According to Proposition 5.1, we can compute $\overline{\phi}(e, G)$ for any $e \in E(G)$ using the following equation:

$$\overline{\phi}(e, G) = \text{degree}(e, G) \qquad (2)$$

It is clear that $\overline{\phi}(e, G)$ can be easily computed with no extra I/O and memory costs. Fig. 4 (b) shows a comparison of $|G|$, $|G_k|$, and $|k$-PG$|$ on the *uk-2005* dataset when k decreases from 100 to 2 in Top-Down. Here, the k-PG is obtained based on the $\overline{\phi}(e)$ in Eq. 2. As shown in the figure, $|G_k|$ decreases as k decreases. For $|k$-PG$|$, it increases as k decreases when k is large. After reaching a peak point with $k = 50$, $|k$-PG$|$ decreases as k decreases. Notably, the peak size of the k-PG is only around 5% of $|G|$, which is much smaller than $|G|$. This indicates that degree(e, G) is a good upper bound of $\phi(e, G)$. In Fig. 4, we use the same notation G_k to denote the input graph before processing a certain connectivity number k for Bottom-Up and Top-Down, but the specific $|G_k|$ values with the same k value for Bottom-Up and Top-Down are different. This is because we process k in different orders: G_k for Bottom-Up is the subgraph constructed by $E_{\phi \geq k}(G)$ while G_k for Top-Down is the subgraph constructed by $E_{\overline{\phi} \leq k}(G)$.

Based on the above discussion, a global upper bound for $\phi(e)$ can already result in a good k-PG in Top-Down. Therefore, to save I/O cost, we will not recompute $\overline{\phi}(e)$ for each k value as we do in the Bottom-Up algorithm.

5.2 The Top-Down Decomposition Algorithm

In this subsection, we focus on how to implement Top-Down in an I/O efficient manner.

Algorithm 5 Top-Down(Graph G)

```
1: compute $\overline{\phi}(e)$ for all $e \in E(G)$;
2: sort all edges $e$ in $E(G)$ on disk by non-increasing order of $\overline{\phi}(e)$;
3: $k = \max_{e \in E(G)}\{\overline{\phi}(e)\}$;
4: $G'_k \leftarrow \emptyset$;
5: while $k > 1$ do
6:     $k$-PG $\leftarrow G'_k$;
7:     for all edge $e$ with $\overline{\phi}(e) = k$ by sequential scanning $G$ on disk do
8:         $E(k$-PG$) \leftarrow E(k$-PG$) \cup \{e\}$ ;
9:     $E_{\phi=k} \leftarrow E($Mem-Decom$(k$-PG$, k))$;
10:    $G'_{k-1} \leftarrow$ CE-Mem$(k$-PG$, E_{\phi=k})$;
11:    $k \leftarrow k - 1$;
```

__A Basic Solution__. Given a graph G, suppose $\overline{\phi}(e)$ has been computed for all $e \in E(G)$, a straightforward solution for Top-Down is to strictly follow the framework in Algorithm 2 as follows: We process k in decreasing order. For each k, we compute the k-PG using RE$(G_k, E_{\overline{\phi}<k}(G_k))$ by scanning G_k once on disk. Then we compute $E_{\phi=k}$ on the k-PG in memory. Lastly, we compute G_{k-1} using CE$(G_k, E_{\phi=k})$ by scanning G_k once again on disk.

__I/O Cost Reduction__. Recall that in our Top-Down algorithm, $G_k =$ CE$(G, E_{\phi>k})$, and we use a global $\overline{\phi}(e)$ for all $e \in E(G)$. Based on this, we can sort all edges $e \in E(G)$ in non-increasing order of $\overline{\phi}(e)$ on disk. It is easy to see that the edges in G_k for each k value are stored sequentially on disk. Therefore, to compute G_k, we do not need to explicitly materialize G_k on disk. On the other hand, if we compute the k-PG using RE$(G_k, E_{\overline{\phi}<k}(G_k))$, we still need to scan G_k once again on disk. To save the I/O cost when computing the k-PG, we can utilize the following proposition:

Proposition 5.2: *Given a graph G, suppose* k-PG $=$ RE(CE$(G,$ $E_{\phi>k}), E_{\overline{\phi}<k})$, *for any $2 \leq k < k_{max}$, we have:*

$$E(k\text{-PG}) = E(\text{CE}((k+1)\text{-PG}, E_{\phi=k+1}) \cup E_{\overline{\phi}=k}. \qquad \square$$

To compute the k-PG by Proposition 5.2, we define a new graph:

$$G'_k = \text{CE}((k+1)\text{-PG}, E_{\phi=k+1}).$$

Suppose we have computed $(k+1)$-PG. We can compute the set $E_{\phi=k+1}$ in the $(k+1)$-PG and compute the graph G'_{k+1} using CE$((k+1)$-PG$, E_{\phi=k+1})$. According to Proposition 5.2, the edges in the k-PG can be computed as $E(k$-PG$) = E(G'_k) \cup E_{\overline{\phi}=k}$. Note that the edges $e \in E(G)$ are sorted in non-increasing order of $\overline{\phi}(e)$, and we process all k values in decreasing order. Therefore, $E_{\overline{\phi}=k}$ can be easily obtained using sequential scan on disk when processing the corresponding k value.

Based on the above discussion, our Top-Down algorithm is shown in Algorithm 5. We first compute $\overline{\phi}(e)$ for all $e \in E(G)$ using Eq. 2 (line 1), and sort all edges $e \in E(G)$ by non-increasing order of $\overline{\phi}(e)$ on disk (line 2). Since k_{max} is unknown, we compute an upper bound of k_{max} as $\overline{k}_{max} = \max_{e \in E(G)}\{\overline{\phi}(e)\}$. We initialize k to be $\overline{k}_{max}$ (line 3) and G'_k to be $\emptyset$ (line 4). Then we process all k values iteratively in decreasing order of k. In each iteration (lines 6-11), we first compute the k-PG using $E(k$-PG$) = E(G'_k) \cup E_{\overline{\phi}=k}$. To do this, we initialize k-PG to be G'_k (line 6) and scan all the edges $e \in E(G)$ with $\overline{\phi}(e) = k$ sequentially on disk (line 7). For each such edge e, we add e into $E(k$-PG$)$ (line 8). After computing the k-PG, we can compute $E_{\phi=k}$ by invoking Mem-Decom$(k$-PG$, k)$ in memory. Lastly, we compute G'_{k-1} using CE$(k$-PG$, E_{\phi=k})$ in memory (line 10) and move to process the next k (line 11). Here, CE-Mem is the in-memory version of the CE-Disk procedure in Algorithm 4 (see Section 4.2).

Example 5.1: Fig. 6 shows a running example of Top-Down on the graph in Fig. 2. The degree number of (v_0, v_2), (v_0, v_3), (v_1, v_3), (v_1, v_4), (v_9, v_{15}), (v_{10}, v_{15}) is 2. The degree number of (v_2, v_7), (v_3, v_7), (v_4, v_7), (v_5, v_7), (v_6, v_7) is 5. The degree number of

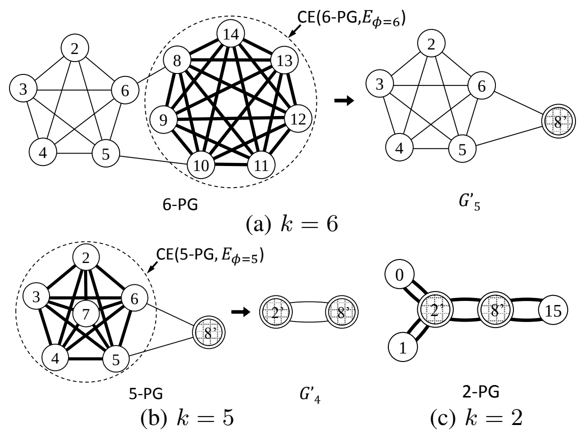


Figure 6: Top-Down Example

(v_8, v_9), (v_8, v_{10}) and (v_9, v_{10}) is 7, and the degree number of the remaining edges is 6. We start from $k = 7$, and 7-PG consists of (v_8, v_9), (v_8, v_{10}), and (v_9, v_{10}), and $E_{\phi=7} = \emptyset$. The 6-PG is shown on the left of Fig. 6 (a). We compute $E_{\phi=6}$, whose edges are shown with bold lines, and contract them. The contracted graph G'_5 is shown in Fig. 6 (a). Then we move to handle $k = 5$. We add the edges with degree$(e) = 5$ and obtain the 5-PG. After computing $E_{\phi=5}$ based on 5-PG, we contract $E_{\phi=5}$ and obtain G'_4 (Fig. 6 (b)). As there are no edges with degree(e) being 4 or 3, $E_{\phi=4} = E_{\phi=3} = \emptyset$ and $G'_4 = G'_3 = G'_2$. When $k = 2$, we obtain 2-PG by adding the edges with degree$(e) = 2$ into G'_2 and compute $E_{\phi=2}$(2-PG) (Fig. 6 (c)). The corresponding $E_{\phi=2} = \{(v_0, v_2), (v_0, v_3), (v_1, v_3), (v_1, v_4), (v_6, v_8), (v_5, v_{10}), (v_9, v_{15}), (v_{10}, v_{15})\}$. □

Complexity Analysis. The peak memory usage and I/O complexity of Top-Down (Algorithm 5) are shown below:

Theorem 5.1: *Given a graph G, let $M^{td}(G)$ be the peak memory used in* Top-Down *(Algorithm 5), we have:*

$$M^{td}(G) = O(\max_{2 \le k \le k_{max}}\{|E_{\phi \le k, \overline{\phi} \ge k}(G)|\}). \quad \square$$

Here, $|E_{\phi \le k, \overline{\phi} \ge k}(G)\}|$ is the size of the k-PG. According to our discussion in Section 5.1, the size of the k-PG is usually much smaller than $|G|$. Therefore, $M^{td}(G)$ is usually much smaller than the memory consumed by the in-memory algorithm.

Theorem 5.2: *Given a graph G, let $I^{bu}(G)$ be the number of I/Os used in* Top-Down *(Algorithm 5), we have:*

$$I^{td}(G) = O(scan(|E(G)|) + sort(|E(G)|)). \quad \square$$

Comparing Theorem 5.2 with Theorem 4.2, the I/O cost of Top-Down to scan edges is smaller than it is for Bottom-Up. However, Top-Down consumes extra I/O cost to sort all edges in G.

Discussion. Algorithm 5 exhibits the worst case behaviour when the input graph is a clique. In this case, the k-PG computed in line 8 is exactly G when $k = k_{max}$. In this case, $M^{td}(G) = O(|E(G)|)$ and $I^{td}(G) = O(scan(|E(G)|) + sort(|E(G)|))$.

6. HYBRID DECOMPOSITION

In this section, we discuss our Hybrid algorithm. As discussed in Section 3.3, Hybrid combines Top-Down and Bottom-Up to seek more opportunities to reduce the size of the k-PG. Hybrid generally follows the Top-Down algorithm, and for each k-PG computed by Top-Down, Hybrid tries to apply the Bottom-Up algorithm to further reduce the size of the k-PG instead of loading the k-PG in memory. Note that according to the discussion in Section 5.1, the k-PG in Top-Down is usually much smaller than G. Therefore, applying Bottom-Up to further reduce the size of the k-PG will not incur much additional I/O cost. On the other hand, as introduced in Section 2, the Mem-Decom algorithm is usually memory intensive. Reducing the size of the k-PG is critical to the scalability of ECC decomposition. Therefore, Hybrid aims to reduce the size of the k-PG without introducing much extra I/O cost.

Algorithm 6 Hybrid(Graph G)

```
1: compute $\overline{\phi}(e)$ for all $e \in E(G)$;
2: sort all edges $e$ in $E(G)$ on disk by non-increasing order of $\overline{\phi}(e)$;
3: $k = \max_{e \in E(G)}\{\overline{\phi}(e)\}$;
4: $G'_k \leftarrow \emptyset$ on disk;
5: while $k > 1$ do
6:     for all edge $e = (u, v)$ with $\overline{\phi}(e) = k$ by sequential scanning $G$ on disk do
7:         add edge $(u, v)$ in $G'_k$ on disk;
8:     compute $k$-PG by invoking Bottom-Up($G'_k$) (Algorithm 4);
9:     $E_{\phi=k} \leftarrow E(k\text{-PG})$
10:    $G'_{k-1} \leftarrow$ CE-Disk($G'_k$, $k$-PG);
11:    $k \leftarrow k - 1$;
```

The Hybrid algorithm is shown in Algorithm 6. The algorithm follows the framework of Algorithm 3. Line 1-3 is the same as Algorithm 5, which computes $\overline{\phi}(e)$ for all $e \in E(G)$, sorts edges according to $\overline{\phi}(e)$, and initializes k. Unlike Algorithm 5, the graph G'_k in Hybrid is stored on disk. The algorithm iteratively processes all k values in decreasing order of k. In each iteration (lines 6-11), the algorithm updates G'_k on disk by adding all edges e with $\overline{\phi}(e) = k$ using sequential scan (lines 6-7). Then, instead of computing $E_{\phi=k}$ on G' directly, the algorithm invokes Bottom-Up(G'_k) (Algorithm 4) to compute the k-PG (line 8) and according to the discussion in Section 3.3, the k-PG contains exactly the set of edges in $E_{\phi=k}$ (line 9). Lastly, the algorithm computes the graph G'_{k-1} on disk by invoking CE-Disk(G'_k,k-PG) (line 10) and moves to process the next k (line 11). The procedure CE-Disk was introduced in Section 4.2.

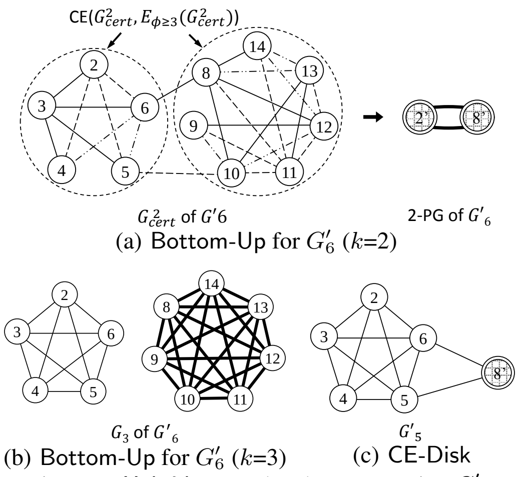

CE(G^2_{cert}, $E_{\phi\ge3}(G^2_{cert})$)
G^2_{cert} of $G'6$
2-PG of G'_6
(a) Bottom-Up for G'_6 (k=2)
G_3 of G'_6
G'_5
(b) Bottom-Up for G'_6 (k=3)
(c) CE-Disk

Figure 7: Hybrid Example when Processing G'_6

Example 6.1: Fig. 7 shows a running example of Hybrid on the graph in Fig. 2. Here, we only show the steps to process G'_6 which is the same as 6-PG in Fig. 6. We invoke Bottom-Up with G'_6 as the input graph. For $k = 2$, we compute the corresponding $G^2_{cert}(G'_6)$ and 2-PG of G'_6, which is shown in Fig. 7 (a). We can then obtain $E_{\phi=2}(G'_6)$. After removing $E_{\phi=2}(G'_6)$, we get G_3 of G'_6, which consists of two separate subgraphs and can be handled individually. We then continue to handle $k = 3, 4, 5$ and obtain $E_{\phi=6}$, whose edges are marked with bold lines in Fig. 7 (b). When $E_{\phi=6}$ is obtained, we contract $E_{\phi=6}$ and obtain G'_5 (Fig. 7 (c)). □

Complexity Analysis. The peak memory usage and I/O complexity of Hybrid (Algorithm 6) are shown below:

Theorem 6.1: *Given a graph G, let $G_k^{hy} = \mathsf{RE}(\mathsf{CE}(G, E_{\phi>k}), E_{\overline{\phi}<k})$, and $M^{hy}(G)$ be the peak memory used in* Hybrid *(Algorithm 6), we have:*

$$M^{hy}(G) = O(\max_{2\leq k\leq k_{max}} M^{bu}(G_k^{hy})). \qquad \Box$$

Here, $M^{bu}(G_k^{hy})$ is the memory used to process G_k^{hy} in Bottom-Up (Algorithm 4). Compared to Theorem 4.1, since $M^{bu}(G_k^{hy}) \leq M^{bu}(G)$, Hybrid outperforms Bottom-Up w.r.t. memory usage. Compared to Theorem 5.1, since $M^{bu}(G_k^{hy}) < O(|G_k^{hy}|)$, Hybrid also outperforms Top-Down w.r.t. memory usage.

Theorem 6.2: *Given a graph G, let $G_k^{hy} = \mathsf{RE}(\mathsf{CE}(G, E_{\phi>k}), E_{\overline{\phi}<k})$, and $I^{hy}(G)$ be the number of I/Os used in* Hybrid *(Algorithm 6), we have:*

$$I^{hy}(G) = O(scan(|E(G)|)+sort(|E(G)|)+\textstyle\sum_{k=2}^{k_{max}} I^{bu}(G_k^{hy})). \qquad \Box$$

Here, $I^{bu}(G_k^{hy})$ is I/O cost to process G_k^{hy} in Bottom-Up (Algorithm 4). Compared to Theorem 5.2, Hybrid consumes an extra I/O cost of $O(\sum_{k=2}^{k_{max}} I^{bu}(G_k^{hy}))$ over Top-Down. However, as discussed in Section 5.1, G_k^{hy} is usually much smaller than graph G. Therefore, the extra I/O cost is usually small.

Discussion. Similar to Top-Down (Algorithm 5), Hybrid (Algorithm 6) exhibits the worst case behaviour when the input graph is a clique. In this case, G'_k computed in line 7 is exactly G when $k = k_{max}$. In this case, $M^{hy}(G) = O(|E(G)|)$ and $I^{hy}(G) = O(k_{max}^2 \cdot scan(|E(G)|) + sort(|E(G)|))$.

7. PERFORMANCE STUDIES

In this section, we present our experimental results. All our experiments are conducted on a machine with an Intel Xeon 2.9 GHz CPU (8 cores) and 32 GB main memory running Linux (Red Hat Enterprise Linux 6.4, 64bit).

Dataset G	**Type**	$\lvert V(G)\rvert$	$\lvert E(G)\rvert$	**Avg Degree**
DBLP	Citation	986,324	6,707,236	13.60
LiveJournal	Social	4,847,571	68,993,773	28.47
Orkut	Social	3,072,441	117,185,083	76.28
uk-2005	Web	39,459,925	936,364,282	47.46
it-2004	Web	41,291,594	1,150,725,436	55.74
twitter-2010	Social	41,652,230	1,468,365,182	70.51
sk-2005	Web	50,636,154	1,949,412,601	76.99

Table 2: Datasets used in Experiments

Datasets. We use seven different types of real-world graphs with different properties for testing (see Table 2). Of these, *LiveJournal* and *Orkut* are downloaded from SNAP (http://snap.stanford.edu/), and the others are downloaded from WEB (http://law.di.unimi.it/).

Algorithms. We implement and compare five algorithms:

- Random-Decom: In-memory algorithm based on [5].
- Exact-Decom: In-memory algorithm based on [9].
- Bottom-Up: Algorithm 4 (Section 4).
- Top-Down: Algorithm 5 (Section 5).
- Hybrid: Algorithm 6 (Section 6).

All algorithms are implemented in C++ and compiled with GNU GCC 4.8.2. Random-Decom and Exact-Decom are the in-memory algorithms used for ECC decomposition by applying the k-ECC computation algorithm in [5] and [9] respectively for all k values. The source code of [5] and [9] was obtained from the authors. A simple heuristic used in [8] is applied in both Random-Decom and Exact-Decom, which computes k-edge connected components in an increasing order of k and takes the k-edge

Graph / Alg	*DBLP*		*LiveJournal*		*Orkut*	
	time	**mem**	**time**	**mem**	**time**	**mem**
Random-Decom	3890s	931.02M	-	-	-	-
Exact-Decom	**18.9**s	636.52M	**1090.8s**	5.98G	1.3 hrs	15.4G
Bottom-Up	38.1s	135.9M	2677.2s	752.5M	4.0 hrs	4.4G
Top-Down	21.9s	66.6M	1451.3s	643.8M	**1.0** hrs	2.0G
Hybrid	22.0s	**66.57**M	1711.5s	**598.7**M	1.1 hrs	**1.9**G

Table 3: Comparison with In-Memory Algorithms

connected components as the input for computing $(k+1)$-edge connected components. In Bottom-Up, Top-Down and Hybrid, we use [9] as Mem-Decom. For each test, we set the maximum running time as 48 hours. For all experiments, we compare the peak memory usage, the total processing time, and the total number of I/Os. However, since the curves for the total number of I/Os are similar to these of the total processing time, we omit the results for the total number of I/Os due to space limitations.

Exp-1: Comparison with In-memory Algorithms. In this experiment, we compare the total processing time and peak memory usage of the five algorithms on three datasets, *DBLP*, *LiveJournal* and *Orkut*. The results are shown in Table 3. If a test can not terminate in the time limit, or fails as a result of out of memory exception, we mark the corresponding cell with '-'.

Generally, the processing time and peak memory usage increase as the size of the graph increases. Random-Decom spends the most time and consumes the most memory of these five algorithms. It can only complete the ECC decomposition on the smallest dataset *DBLP*. The reason for Random-Decom's long processing time is the large number of iterations involved, which is the fundamental step of [5], during processing.

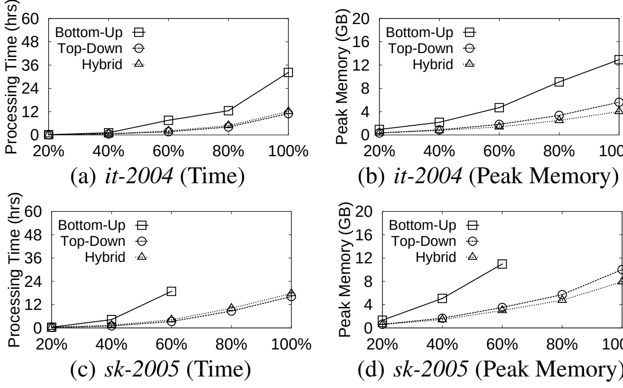
For the remaining four algorithms, Exact-Decom consumes much more memory than our proposed algorithms. For example, on *Orkut*, it consumes 3.5, 7.7, and 8.1 times more memory than Bottom-Up, Top-Down and Hybrid respectively. This is because Exact-Decom keeps the whole graph in memory during processing. Top-Down runs fastest among our proposed algorithms. This is because apart from sorting the input graph once, Top-Down only scans the input graph once in total. The processing time of Hybrid is close to Top-Down (18% more on *LiveJournal* and 10% more on *Orkut*), and Hybrid consumes the least memory. The reason for this is that Hybrid uses Bottom-Up to reduce peak memory usage. On *DBLP*, Hybrid does not show significant improvement, since the memory usage of Top-Down is already very small. Bottom-Up takes less memory than Exact-Decom because the size of G_{cert}^k and k-PG used in Bottom-Up is much smaller than $|G|$. Of our proposed algorithms, however, it takes the most time and memory on these three datasets. This is because Bottom-Up needs to scan G_k multiple times for a certain k, and the size of G_{cert}^k is usually bigger than the k-PG used in Top-Down and Hybrid. Remarkably, on *Orkut*, Top-Down and Hybrid outperform Exact-Decom on processing time (1.0 hours, 1.1 hours and 1.3 hours respectively). This is the result of the cost sharing technique used in our proposed algorithms to reduce redundant computations.

Exp-2: Performance on Big Graphs. In this experiment, we compare the total processing time and peak memory usage of our proposed algorithms on four big real datasets: *uk-2005*, *it-2004*, *twitter-2010* and *sk-2005*. The results are shown in Table 4. Since both Random-Decom and Exact-Decom run out of memory on all four big graphs, we only compare our proposed algorithms.

On these four datasets, Top-Down runs fastest and the processing time of Hybrid is close to Top-Down. However, compared with the saved memory, the extra time cost for Hybrid is usually small. For example, on the largest dataset *sk-2005*, Hybrid takes 9.6% more time than Top-Down but consumes 21% less memory

Graph / Alg	*uk-2005*		*it-2004*		*twitter-2010*		*sk-2005*	
	time	**mem**	**time**	**mem**	**time**	**mem**	**time**	**mem**
Bottom-Up	15.56 hrs	9.34G	32.17 hrs	12.93G	-	-	-	-
Top-Down	5.90 hrs	3.45G	11.01 hrs	5.62G	34.87 hrs	7.22G	16.17 hrs	10.03G
Hybrid	6.52 hrs	2.97G	12.06 hrs	4.03G	35.01 hrs	6.81G	17.73 hrs	7.92G

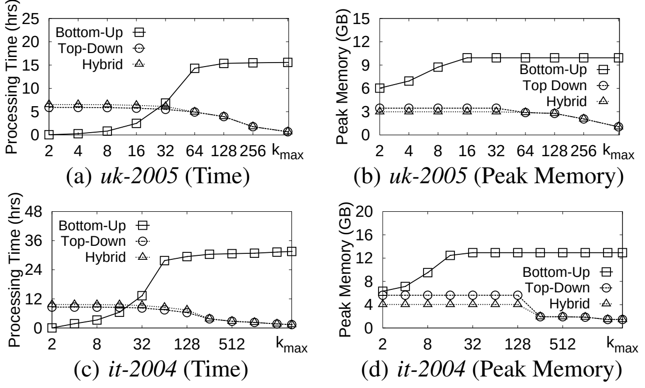
Table 4: Performance on Big Graphs



(a) *it-2004* (Time) (b) *it-2004* (Peak Memory)

(c) *sk-2005* (Time) (d) *sk-2005* (Peak Memory)

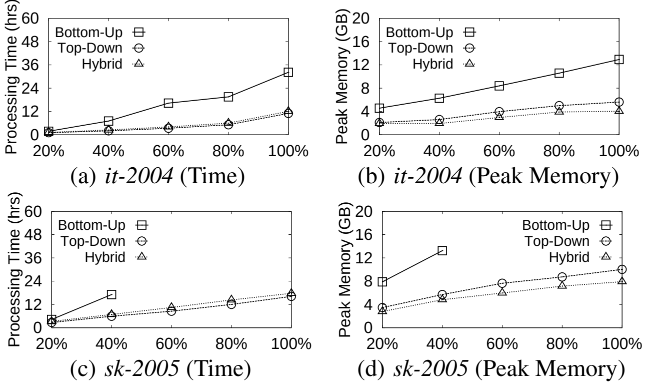
Figure 8: Vary $|V|$ (Scalability)



(a) *uk-2005* (Time) (b) *uk-2005* (Peak Memory)

(c) *it-2004* (Time) (d) *it-2004* (Peak Memory)

Figure 10: Performance for Each k



(a) *it-2004* (Time) (b) *it-2004* (Peak Memory)

(c) *sk-2005* (Time) (d) *sk-2005* (Peak Memory)

Figure 9: Vary $|E|$ (Scalability)

than Top-Down. Of the three algorithms, Bottom-Up takes more processing time and memory than the other two. For example, on *uk-2005*, the processing time and peak memory usage of Bottom-Up are respectively 2.64 and 2.70 times more than Top-Down, and Bottom-Up cannot finish the decomposition on *twitter-2010* and *sk-2005*. Note that although Bottom-Up is slower and consumes more memory than Top-Down and Hybrid, it is still useful for the following two reasons: First, Bottom-Up is used as a subroutine of Hybrid, and by exploiting Bottom-Up, Hybrid consumes less memory than Top-Down, as shown in Table 4. Second, in some applications, such as [19], a user may be interested in the k-ECCs with a small k, for example, $k \leq 5$. Bottom-Up is very suitable for these applications whereas Top-Down and Hybrid need to explore all the possible k values from k_{max} to 2 to compute these k-ECCs.

Exp-3: Scalability Testing. We vary $|V|$ and $|E|$ from 20% to 100% of two large datasets *it-2004* and *sk-2005* and test the scalability of our proposed algorithms. The results are shown in Fig. 8 and Fig. 9.

As shown in Fig. 8, both the processing time and peak memory usage increase for our proposed algorithms when $|V|$ increases. This is because as $|V|$ increases, the maximum size of k-PG (and also G^k_{cert} for Bottom-Up) for each algorithm also increases. Of all the algorithms, Bottom-Up consumes the most time and memory while Top-Down takes the least processing time and Hybrid consumes the least memory, which is consistent with our complexity analysis. In Fig. 8 (a) and (c), the gap in processing time between Top-Down and Hybrid remains stable as $|V|$ increases, while the gap in peak memory usage increases more sharply as $|V|$ increases (Fig. 8 (b) and (d)). This is because, for Top-Down, as $|V|$ increases, the number of edges with $\overline{\phi}(e) \geq k$ and $\phi(e) < k$ for each k-PG also increases. Hybrid eliminates this kind of edges and obtains a smaller k-PG without much extra cost. In Fig. 8 (a) and (b), Bottom-Up takes much more processing time and memory than Top-Down and Hybrid, and Fig. 8 (c) and (d) show that when $|V| > 60\%$, Bottom-Up cannot finish the decomposition.

Fig. 9 shows that, when $|E|$ increases, both the processing time and peak memory usage increase for all algorithms. For Top-Down and Hybrid, the processing time on *it-2004* (Fig. 9 (a)) and *sk-2005* (Fig. 9 (c)) is very close while the difference in peak memory usage increases as $|E|$ increases (Fig. 9 (b) and (d)). This is because Hybrid can obtain a smaller k-PG by eliminating the edges with $\overline{\phi}(e) \geq k$ and $\phi(e) < k$. Bottom-Up takes the most processing time and memory and cannot finish the decomposition when $|E| > 40\%$ on *sk-2005* (Fig. 9 (c) and (d)).

Exp-4: Performance for Each k. In this experiment, we compare the cumulative processing time and peak memory usage as k increases for Bottom-Up, and decreases for Top-Down and Hybrid on *uk-2005* and *it-2004*. The results are shown in Fig. 10.

Fig. 10 (a) shows that for Bottom-Up, as k increases, the processing time grows sharply at first (from $k = 2$ to $k = 64$), and then keeps stable (from $k = 64$ to $k = k_{max}$). This is because initially, G_k is too large to be processed in memory and Bottom-Up needs to scan G_k on disk to compute G^k_{cert} and k-PG; as k increases, more edges with $\phi(e) < k$ are removed and G_k can be processed in memory. All the operations are then performed in memory. For the same reason, the processing time of Bottom-Up demonstrates a similar trend on *it-2004* (Fig. 10 (c)). For the peak memory usage, in Fig. 10 (b), as k increases, it increases and reaches the peak point when $k = 16$. Thereafter, it remains unchanged. This is because the maximum size of G^k_{cert} usually determines the peak memory usage of Bottom-Up. According to Proposition 4.2, $E(G^k_{cert}) \leq (k+1) \times (|V(G_k)| - 1)$, therefore, when k is small, $|V(G_k)|$ is large and the decreasing rate of $|V(G_k)|$ is slower than the increasing rate of k. As a result, the size of G^k_{cert} increases. At some certain k, G^k_{cert} reaches the peak point and after that, although k still increases, $|V(G_k)|$ becomes small and the peak size of G^k_{cert} remains unchanged. Bottom-Up has a similar trend on *it-2004* (Fig. 10 (d)).

Fig. 10 (a) shows that for Top-Down, the processing time remains stable at first (from k_{max} to 256) and then grows fast (from 256 to 2) as k decreases. The reason is that the degree of the graph follows a power-law distribution and the edges with $2 \leq \mathsf{degree}(e) \leq 256$ constitute the majority of the edges of the graph. Therefore, the size of corresponding k-PG with $2 \leq k \leq 256$ is also large. Consequently, the cumulative processing time grows fast when k decreases from 256 to 2. As the size of k-PG also determines the peak memory usage of Top-Down, peak memory also remains stable when k decreases from k_{max} to 256 and then grows fast when k decreases from 256 to 2, as shown in Fig. 10 (b). We make a similar observation on *it-2004* for processing time (Fig. 10(c)) and peak memory usage (Fig. 10(d)). As Hybrid follows a similar framework to Top-Down, it exhibits similar trends to Top-Down in Fig. 10. The effectiveness of reducing Hybrid's memory is evident when k becomes small. For example, it is evident when $k < 64$ in Fig. 10 (b) and when $k < 256$ in Fig. 10 (d), while the corresponding processing time is close to that of Top-Down in Fig. 10 (a) and Fig. 10 (c).

8. RELATED WORK

We review the related work from two categories, namely, cohesive subgraph models and I/O efficient graph algorithms.

Cohesive Subgraph Models. Cohesive subgraph computation is an important problem in network analysis and there are many different models of cohesive subgraph in the literature. One of the earliest graph models is the clique model [18]. As clique is often too restrictive for many applications, more clique relaxation models have been proposed. The n-clique model [17] requires the distance between any two nodes in the subgraph to be at most n. The quasi-clique model can be either a relaxation on the density [1] or the degree [21]. Other models are also studied in the literature. k-core [22] is the largest subgraph of a graph in which the degree of each node is at least k. The k-truss [15] model, triangle k-core [27] model and DN-Graph [24] model are defined based on triangles. A k-mutual-friend subgraph model is introduced in [31]. k-edge connected component computation is studied in [26, 32, 5, 9].

I/O Efficient Graph Algorithms. With the increase in graph size, several graph algorithms focusing on I/O efficiency have been proposed in the literature. In [11], Cheng et al. describe an I/O efficient algorithm for the core decomposition problem in massive networks. Zhang et al.[29] study an I/O efficient algorithm to compute the strongly connected components in a graph in the semi-external model and extend the algorithm to the external memory model in [28]. I/O efficient algorithms for the triangle enumeration problem are presented in [14, 16]. And I/O efficient algorithms for the maximal clique enumeration problem are proposed in [12, 13]. The I/O efficient algorithm for the k-truss problem is investigated in [23]. A connectivity index for massive-disk resident graphs is studied in [3]. An I/O efficient semi-external algorithm for the depth first search is proposed in [30].

9. CONCLUSION

In this paper, we study the problem of ECC graph decomposition. We propose I/O efficient techniques to reduce the size of the graph to be loaded into memory and explore possible cost sharing when computing k-ECCs for different k values. We introduce two elegant graph reduction operators to reduce the memory size and three novel algorithms to reduce the CPU and I/O costs. We conduct extensive experiments using seven real datasets to demonstrate the efficiency of our approach.

Acknowledgements. Lu Qin is supported by ARC DE140100999 and ARC DP160101513. Xuemin Lin is supported by NSFC61232006, ARC DP140103578 and ARC DP150102728. Lijun Chang is supported by ARC DE150100563 and ARC DP160101513. Wenjie Zhang is supported by ARC DP150103071 and ARC DP150102728.

10. REFERENCES

[1] J. Abello, M. G. Resende, and S. Sudarsky. Massive quasi-clique detection. In *LATIN 2002: Theoretical Informatics*, pages 598–612. 2002.

[2] A. Aggarwal, J. Vitter, et al. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[3] C. Aggarwal, Y. Xie, and P. S. Yu. Gconnect: A connectivity index for massive disk-resident graphs. *PVLDB*, 2(1):862–873, 2009.

[4] R. Agrawal, S. Rajagopalan, R. Srikant, and Y. Xu. Mining newsgroups using networks arising from social behavior. In *Proc. of WWW'03*, pages 529–535, 2003.

[5] T. Akiba, Y. Iwata, and Y. Yoshida. Linear-time enumeration of maximal k-edge-connected subgraphs in large networks by random contraction. In *Proc. of CIKM'13*, pages 909–918, 2013.

[6] J. I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani. K-core decomposition of internet graphs: Hierarchies, self-similarity and measurement biases. *Networks and Heterogeneous Media*, 3(2), 2008.

[7] S. Carmi, S. Havlin, S. Kirkpatrick, Y. Shavitt, and E. Shir. A model of internet topology using k-shell decomposition. *Proceedings of the National Academy of Sciences*, 104(27):11150–11154, 2007.

[8] L. Chang, X. Lin, L. Qin, J. X. Yu, and W. Zhang. Index-based optimal algorithms for computing Steiner components with maximum connectivity. In *Proc. of SIGMOD'15*, pages 459–474, 2015.

[9] L. Chang, J. X. Yu, L. Qin, X. Lin, C. Liu, and W. Liang. Efficiently computing k-edge connected components via graph decomposition. In *Proc. of SIGMOD'13*, pages 205–216, 2013.

[10] J. Chen and B. Yuan. Detecting functional modules in the yeast protein–protein interaction network. *Bioinformatics*, 22(18):2283–2290, 2006.

[11] J. Cheng, Y. Ke, S. Chu, and M. T. Ozsu. Efficient core decomposition in massive networks. In *Proc. of ICDE'11*, pages 51–62, 2011.

[12] J. Cheng, Y. Ke, A. W.-C. Fu, J. X. Yu, and L. Zhu. Finding maximal cliques in massive networks. *ACM Transactions on Database Systems*, 36(4):21, 2011.

[13] J. Cheng, L. Zhu, and Y. Ke. Fast algorithms for maximal clique enumeration with limited memory. In *Proc. of SIGKDD'12*, pages 1240–1248, 2012.

[14] S. Chu and J. Cheng. Triangle listing in massive networks. *ACM Transactions on Knowledge Discovery from Data*, 6(4):17, 2012.

[15] J. Cohen. Trusses: Cohesive subgraphs for social network analysis. *National Security Agency Technical Report*, page 16, 2008.

[16] X. Hu, Y. Tao, and C. Chung. Massive graph triangulation. In *Proc. of SIGMOD'13*, pages 325–336, 2013.

[17] R. D. Luce. Connectivity and generalized cliques in sociometric group structure. *Psychometrika*, 15(2):169–190, 1950.

[18] R. D. Luce and A. D. Perry. A method of matrix analysis of group structure. *Psychometrika*, 14(2):95–116, 1949.

[19] T. L. Magnanti and S. Raghavan. Strong formulations for network design problems with connectivity requirements. *Networks*, 45(2), 2005.

[20] H. Nagamochi and T. Ibaraki. A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica*, 1992.

[21] J. Pei, D. Jiang, and A. Zhang. On mining cross-graph quasi-cliques. In *Proc. of SIGKDD'05*, pages 228–238, 2005.

[22] S. B. Seidman. Network structure and minimum degree. *Social Networks*, 5(3):269–287, 1983.

[23] J. Wang and J. Cheng. Truss decomposition in massive networks. *PVLDB*, 5(9):812–823, 2012.

[24] N. Wang, J. Zhang, K. Tan, and A. K. H. Tung. On triangulation-based dense neighborhood graphs discovery. *PVLDB*, 4(2):58–68, 2010.

[25] D. R. White and F. Harary. The cohesiveness of blocks in social networks: Node connectivity and conditional density. *Sociological Methodology*, 31(1), 2001.

[26] X. Yan, X. Zhou, and J. Han. Mining closed relational graphs with connectivity constraints. In *Proc. of SIGKDD'05*, pages 324–333, 2005.

[27] Y. Zhang and S. Parthasarathy. Extracting analyzing and visualizing triangle k-core motifs within networks. In *Proc. of ICDE'12*, pages 1049–1060, 2012.

[28] Z. Zhang, L. Qin, and J. X. Yu. Contract & expand: I/O efficient SCCs computing. In *Proc. of ICDE'14*, pages 208–219, 2014.

[29] Z. Zhang, J. X. Yu, L. Qin, L. Chang, and X. Lin. I/O efficient: Computing SCCs in massive graphs. In *Proc. of SIGMOD'13*, pages 245–270, 2013.

[30] Z. Zhang, J. X. Yu, L. Qin, and Z. Shang. Divide & conquer: I/O efficient depth-first search. In *Proc. of SIGMOD'15*, pages 445–458, 2015.

[31] F. Zhao and A. K. Tung. Large scale cohesive subgraphs discovery for social network visual analysis. *PVLDB*, 6(2):85–96, 2012.

[32] R. Zhou, C. Liu, J. X. Yu, and W. Liang. Finding maximal k-edge-connected subgraphs from a large graph. In *Proc. of EDBT'12*, pages 480–491, 2012.