Efficiently Computing k-Edge Connected Components via Graph Decomposition

Lijun Chang[†£], Jeffrey Xu Yu[‡], Lu Qin[‡], Xuemin Lin[†£], Chengfei Liu[§], Weifa Liang[¶]

[†]East China Normal University, China
[£]University of New South Wales, Australia, {ljchang,lxue}@cse.unsw.edu.au
[‡]The Chinese University of Hong Kong, China, {yu,lqin}@se.cuhk.edu.hk
[§]Swinburne University of Technology, Australia, cliu@swin.edu.au
[¶]Australian National University, Australia, wliang@cs.anu.edu.au

ABSTRACT

Efficiently computing k-edge connected components in a large graph, $G = (V, E)$, where V is the vertex set and E is the edge set, is a long standing research problem. It is not only fundamental in graph analysis but also crucial in graph search optimization algorithms. Consider existing techniques for computing k-edge connected components are quite time consuming and are unlikely to be scalable for large scale graphs, in this paper we firstly propose a novel graph decomposition paradigm to iteratively decompose a graph G for computing its k-edge connected components such that the number of drilling-down iterations h is bounded by the "depth" of the k-edge connected components nested together to form G, where h usually is a small integer in practice. Secondly, we devise a novel, efficient threshold-based graph decomposition algorithm, with time complexity $O(l \times |E|)$, to decompose a graph G at each iteration, where l usually is a small integer with $l \ll |V|$. As a result, our algorithm for computing k-edge connected components significantly improves the time complexity of an existing state-of-the-art technique from $O(|V|^2|E| + |V|^3 \log |V|)$ to $O(h \times l \times |E|)$. Finally, we conduct extensive performance studies on large real and synthetic graphs. The performance studies demonstrate that our techniques significantly outperform the state-of-the-art solution by several orders of magnitude.

Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Data mining; G.2.2 [**Graph Theory**]: Graph algorithms

Keywords

k-edge connected components; graph decomposition; minimum cut

1. INTRODUCTION

In many real applications, various data and their relationships can be modeled as a graph $G = (V, E)$, where vertices in V represent the entities of interest and edges in E represent the relationships between the entities. With the proliferation of graph data based applications, such as social networks, information networks, web search, bi-chemistry, biology, and road networks, significant research effort has been made towards many fundamental problems in managing and analyzing graph data. Given a graph $G = (V, E)$, the problem of computing k-edge connected components of G is to find a collection of maximal induced subgraphs $\{G_i\}$ of G such that each G_i is k-edge connected, i.e., the resulting graph is still connected after the removal of any $(k-1)$ edges from G_i [3, 20]. Fig. 1 shows an example; the graph in Fig. 1 is 2-edge connected, and it has four 3-edge connected components highlighted by the dotted circles, respectively.

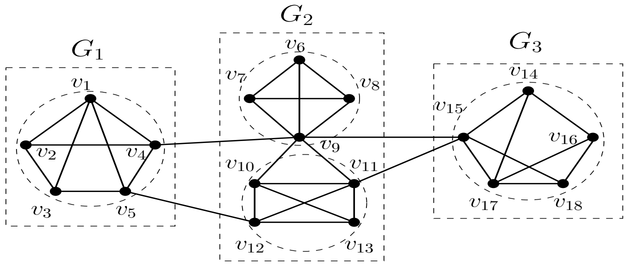


Figure 1: 3-edge connected components

Computing k-edge connected components has many real applications. For example, in social networks, computing k-edge connected components can identify the closely related entities to provide useful information for social behavior mining [1]. In a web-link based graph, a highly connected graph may be a group of web pages with a high commonality, which is useful for identifying the similarities among web pages. In computational biology, a highly connected subgraph is likely to be a functional cluster of genes for biologists to conduct the study of gene microarrays. Computing k-edge connected components also potentially contributes to many other technology developments such as graph visualization, robust detection of communication networks, community detection in a social network (e.g., Facebook), graph sparsification, etc, [7, 16, 19, 20]. Motivated by these, the problem of computing k-edge connected components has been studied very recently in [17, 20].

Clearly, if a graph G is not k-edge connected, there must be a set C of edges, namely a cut, such that the number $|C|$ of edges in C is smaller than k (i.e., $|C| < k$) and the removal of the edges in C cuts the graph G into two disconnected subgraphs G_1 and G_2. Based on this observation, the authors in [17, 20] develop a cut-based technique to iteratively cut a non k-edge connected graph into two disconnected subgraphs by applying the *global min-cut* algorithm in [14] (to be defined in the next section) till all remaining connected subgraphs are k-edge connected or no edge is remaining in the graph. For example, considering the graph in Fig. 1 and a given $k = 3$, suppose that a cut $\{(v_4, v_9), (v_5, v_{12})\}$ is first discovered. The cut-based technique in [17, 20] cuts the graph into G_1 and $(G_2 \cup G_3)$ where G_1 is induced by vertices v_1, v_2, $\ldots$, v_5, and $(G_2 \cup G_3)$ is induced by v_6, $\ldots$, v_{18}. In the next iteration, the cut-based technique is executed against subgraphs G_1 and $(G_2 \cup G_3)$, respectively. Since the time complexity of the technique in [14] is $O(|V||E| + |V|^2 \log |V|)$, the cut-based technique runs in $O(|V||E| + |V|^2 \log |V|)$ time to find one cut. Consequently, the cut-based technique in [17, 20] runs in time $O(|V|^2(|E| + |V| \log |V|))$ to compute k-edge connected components of G. The authors in [20] also develop a set of novel techniques to reduce the running time of the proposed algorithm in practice. Nevertheless, due to its high order time complexity and the nature of the framework (cut-based), the techniques and the framework are not scalable to efficiently process large scale graphs; this is also confirmed by our empirical studies in Section 5. Motivated by this, in this paper we study the problem of efficiently computing k-edge connected components.

Our Approach. Our technique is based on a graph decomposition paradigm. We here use the graph G in Fig. 1 as an illustrative example, which consists of three subgraphs G_1, G_2, and G_3, as depicted by dotted rectangles, such that G_1 is connected to the other part of G by two edges, and G_3 is also connected to the other part of G by two edges. Assuming that $k = 3$, we can decompose the graph into three disconnected subgraphs G_1, G_2, and G_3 instead of separating G_1 from G.

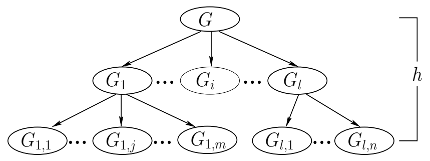


Figure 2: Graph decomposition tree

Generally, a non k-edge connected ($k > 1$) graph G can be represented by a decomposition tree as depicted in Fig. 2 such that each k-edge connected component of G is represented by a leaf node, and each intermediate node of the tree is a connected subgraph of G that is not k-edge connected. Moreover, the sibling nodes (subgraphs) in the tree have the property that each of them is connected to the remaining part of the parent graph by less than k edges except one sibling subgraph. For example, in Fig. 2, G_1, G_2, ..., G_l are the sibling graphs with the parent graph G, and there are at least $(l - 1)$ graphs among G_1, G_2, ..., G_l each of which (say G_i) is connected to the remaining part of G (i.e., $G - G_i$) by less than k edges. Our graph decomposition paradigm guarantees to decompose an intermediate graph G by detecting and removing all inter edges cross its children subgraphs.

To efficiently compute k-edge connected components of a graph using the graph decomposition paradigm, we develop a novel, efficient algorithm to decompose graph G for a given k, which runs in time $O(l|E|)$ and has the following property that 1) if G is not k-edge connected, it will remove all the edges cross its children subgraphs (in the decomposition tree), and 2) each k-edge connected component will not be broken into several parts. Iteratively running graph decomposition algorithm (till all remaining connected subgraphs are k-edge connected) gives a solution to the problem of computing k-edge connected components; this gives the overall time complexity $O(h \times l|E|)$ instead of $O(|V|^2(|E| + |V| \log |V|))$, where h is the height of the decomposition tree (see Fig. 2) and $l \ll |V|$.

Contributions. Our primary contributions are summarized as follows.

- We propose an effective graph decomposition paradigm to efficiently compute k-edge connected components.
- We develop a novel graph decomposition technique to efficiently decompose a non k-edge connected graph in $O(l \times |E|)$ time, where l usually is a small integer with $l \ll |V|$. Consequently, our technique for computing k-edge connected components runs in $O(h \times l \times |E|)$ instead of $O(|V|^2(|E| + |V| \log |V|))$ time of the state-of-the-art technique in [20], where h is the height of the decomposition tree and is practically a small integer.
- We conduct both theoretical and empirical studies on large real and synthetic graphs. Extensive performance studies demonstrate that the proposed algorithm significantly outperforms the state-of-the-art algorithm by several orders of magnitude.

Organization. The rest of the paper is organized as follows. A brief overview of related work is given below. Section 2 provides the necessary background information, including the problem definition, the global min-cut algorithm in [14], and an existing algorithm for testing edge connectivity [11]. Section 3 presents the graph decomposition paradigm, while Section 4 presents the graph decomposition technique. The experimental results are reported in Section 5. Finally, a conclusion is given in Section 6.

Related Work. Efficiently computing subgraphs, based on some designated densities, has recently drawn a great deal of attentions from both database and algorithm communities. The designated densities can be classified into two categories, 1) local density, and 2) connectivity density.

1. Local Density. Efficient techniques for computing all maximal *cliques* and *quasi-cliques* of a graph are presented in [6, 4] and [18], respectively. The authors in [16] investigate the problem of efficiently enumerating another kind of dense induced subgraphs, namely DN-subgraphs, where a DN-subgraph has the property that for each pair of adjacent vertices, u and v, they share at least λ common neighbors. Problems of efficiently computing other dense subgraphs, including k-core [5], triangle k-core motifs [19], etc., have also been recently investigated. An extension of k-core, i.e., finding dense subgraphs g of each graph G such that the h-hop neighborhood of a vertex in g has at least k vertices whose attributes satisfy a certain constraint, is also studied in [9]. Nevertheless, due to inherently different problem natures, these existing techniques are inapplicable to computing k-edge connected components.

2. Connectivity Density. The authors in [17] investigate the problem of efficiently computing frequent closed k-edge connected subgraphs from a set of data graphs with the focus on k-edge connectivity and the number of occurrence of subgraphs. The edge connectivity of a graph is the minimum number of edges whose removal disconnects the graph. Note that a frequent closed subgraph is not necessarily an induced subgraph. The only existing technique to tackle the scalability of computing k-edge connected components is reported in [20]. The technique is based on iteratively cutting a graph with lower (than k) connectivity into two parts, and the authors in [20] also develop a set of novel pruning rules to reduce the running time in practice. Similar (to that in [20]) framework is also used to compute SkyGraph in [13]. Here, the problem of computing SkyGraph outputs a subset $\mathcal{S}$ of induced subgraphs of G with the properties: 1) for each induced subgraph $g \notin \mathcal{S}$, there must be a subgraph g' in $\mathcal{S}$ such that both the connectivity of g' and the number of vertices of g' are no smaller than those of g with at least one of these two strictly larger; and 2) none of two subgraphs in $\mathcal{S}$ has such property regarding each other. As pointed out earlier, due to the high order time complexity and the nature of the framework (cut-based), the techniques and the framework in [20] are not scalable enough to efficiently process large scale graphs.

The problem of efficiently computing every maximal subset V_i of vertices of G, such that each pair of vertices in V_i is k-edge connected in G, has been studied in [8, 12, 10, 15]. Note that in such a maximal vertex subset V_i of G, each pair of vertices that are k-edge connected in G is not necessarily k-edge connected in the induced subgraph by V_i; in fact, the induced subgraph may be disconnected. Therefore, computing all such maximal vertex subsets is inherently different from computing the k-edge connected components of G. Note that, although [10] also uses the term "k-edge connected component", the meaning is different as discussed above.

2. BACKGROUND INFORMATION

In this paper, we focus on an *undirected graph* $G = (V, E)$, where V is the set of vertices and E is the set of edges. We denote the number of vertices and the number of edges by $|V|$ and $|E|$, respectively.

Given a vertex subset $V_s \subseteq V$, the *induced* subgraph $G[V_s]$ by the vertices in V_s is a subgraph of G with V_s as its vertex set such that the edge set of $G[V_s]$ consists of only the edges in G with both endpoints in V_s. That is, $G[V_s] = (V_s, \{(u, v) \in E \mid u, v \in V_s\})$.

2.1 Problem Statement

Definition 2.1: A graph G is *k-edge connected* if the remaining graph is still connected after the removal of any $(k - 1)$ edges from G. □

For example, the graph G in Fig. 3 is 2-edge connected, while the two subgraphs G_1 and G_3 in Fig. 3 are 3-edge connected.

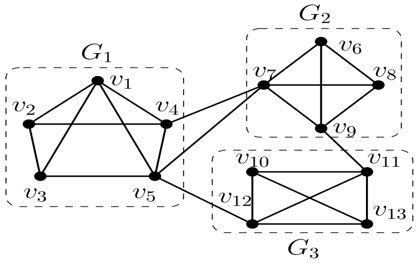


Figure 3: A graph and its 3-edge connected components

Definition 2.2: Given a graph G, a subgraph g of G is a *k-edge connected component* if 1) g is k-edge connected, and 2) any supergraph of g in G is not k-edge connected. □

Problem Statement: Given a graph G and an integer k, we study the problem of efficiently computing k-edge connected components of G.

Properties of k-edge connected components:

- A k-edge connected component is an *induced* subgraph;
- A k-edge connected component is *maximal*; that is, by adding any vertex or a set of vertices into a k-edge connected component, the resulting induced subgraph will not be k-edge connected;
- All k-edge connected components of a graph are *disjoint*.

2.2 Cut-based Framework, Global Min-Cut, and Connectivity Test

Definition 2.3: Given a graph $G = (V, E)$, a *cut* $C = (S, T)$ is a partition of V into two non-empty, disjoint subsets, $S \cup T = V$ and $S \cap T = \emptyset$. □

We also denote a cut by the set of edges whose endpoints are in different subsets, i.e., $\{(u, v) \in E \mid u \in S, v \in T\}$. The value of a cut is the number of edges in the cut, i.e., $w(C) = w(S, T) = |\{(u, v) \in E \mid u \in S, v \in T\}|$.

Definition 2.4: A cut $C = (S, T)$ is called an *s–t cut* if s and t are in different partitions, and it is a *minimum s–t cut* if its value is no larger than the value of any other s–t cuts. □

Let $\lambda(s, t; G)$ denote the value of a minimum s–t cut in G. The *connectivity* between s and t in G is defined as $\lambda(s, t; G)$. Two vertices s and t are called k-edge connected in G if and only if $\lambda(s, t; G) \geq k$. In the following, we simply denote k-edge connected as *k-connected*.

Definition 2.5: The *global min-cut* of a graph G is the cut of G that has the smallest value among all cuts of G. □

Let $\lambda(G)$ denote the value of the global min-cut of G, or equivalently, $\lambda(G) = \min_{s,t \in G, s \neq t} \lambda(s, t; G)$. A graph G is k-connected if and only if $\lambda(G) \geq k$. For example, $C_1 = \{(v_4, v_7), (v_5, v_7), (v_5, v_{12})\}$ and $C_2 = \{(v_5, v_{12}), (v_9, v_{11})\}$ are cuts of the graph in Fig. 3. C_1 is a minimum v_1–v_8 cut and C_2 is a global min-cut.

Cut-based framework: For computing k-edge connected components of a graph G, an existing solution [20] uses a cut-based framework by iteratively computing a small cut of each connected subgraph by running a variant of the global min-cut algorithm, and removing all edges in cuts with values less than k. The connected subgraphs in the final graph are k-edge connected components of G. The pseudocode is shown below.

```
1: Procedure: find $k$-edge connected components ($G$, $k$)
2: Find a small cut, $C$, of $G$;
3: if the value of $C$ is less than $k$ then
4:     Remove all edges of $C$ from $G$;
5:     Find $k$-edge connected components of each connected subgraph of $G$;
6: else
7:     Output $G$ as a $k$-edge connected component;
```

Computing a global min-cut: We introduce an approach to finding global min-cut of graphs [14]. The general idea is finding minimum s–t cuts for $(|V| - 1)$ pairs of vertices, and reporting the one with the smallest value as a global min-cut. Instead of computing maximum flows, the authors in [14] propose a procedure called the *maximum adjacency search* or *maximum cardinality search*, denoted by Mas, to find a minimum s–t cut. Given a graph as input, Mas returns the minimum cut for a pair of vertices s and t. The efficiency of this approach is due to the fact that s and t are determined by Mas rather than its input. Whenever a minimum s–t cut is found by Mas, s and t are merged into a super-vertex, and the resulting graph is given as an input to Mas for another iteration. The global min-cut is found after $(n - 1)$ iterations.

The procedure Mas computes an order of all vertices in G, denoted by a list L. Let t be the last vertex in L and s be the vertex prior to t in L. Then it has the property that the adjacent edges of t in G is the minimum s–t cut. The list L is constructed as follows. It is initialized as a singleton list containing an arbitrary vertex from V. As long as there are vertices not included in L, the vertex u, which is the one most tightly connected to L, i.e., $u = \arg\max_{v \in V \setminus L} w(L, v)$, where $w(L, v)$ denotes the number of edges between v and the vertices in L, is added to the tail of L.

Theorem 2.1: *[14] Let s and t be the two vertices (in the order) most recently added to L, then $(L \setminus \{t\}, \{t\})$ is a minimum s–t cut. The time complexity of* Mas *is $O(|E| + |V| \log |V|)$ if the Fibonacci heap is used for finding the most tightly connected vertex.* □

As a consequence of Theorem 2.1, the time complexity of finding a global min-cut of G is $O(|V||E| + |V|^2 \log |V|)$ [14]. In [20], the authors propose that, in order to compute k-edge connected components, as long as the value of the minimum cut found by Mas in an iteration is smaller than k, the procedure for finding a small cut will terminate. Note that the time complexity of Mas is $\Omega(|E|)$, i.e., it is lower bounded by $|E|$. Therefore, if a graph is k-connected, the running time of [20] is $\Omega(|V||E|)$ which limits the algorithm to be only applicable to small graphs.

Testing edge connectivity: The problem of testing whether a graph is k-edge connected is also studied in the literature. For unweighted simple graphs, the best previous known algorithm runs in time $O(|V||E|)$ which is based on dominating set and maximum flow techniques [11], denote by TestConnect. If the minimum degree of a graph G is less than k, then G is not k-edge connected. Otherwise, the algorithm maintains three disjoint vertex subsets S, T, U of V, where T consists of all vertices adjacent to but not in S, and U consists of all remaining vertices. S is initialized to contain an arbitrary vertex. Whenever U is non-empty, it picks a vertex v from U and computes the minimum cut C between v and S. If the value of C is less than k, then G is not k-edge connected, otherwise, v is added to S, and T and U are updated accordingly. Finally if U is empty, then G is k-edge connected. For more details, please refer to [11]. Note that, TestConnect terminates when a cut of value less than k is found or U is empty, and with a slight modification it can be used in the cut-based framework to replace the procedure of finding global min-cut. However, existing works for computing k-edge connected components of a graph are unaware of this algorithm. We also evaluate the performance of using TestConnect in our empirical studies (see Section 5).

3. COMPUTING K-EDGE CONNECTED COMPONENTS

Before presenting techniques to compute k-edge connected components, we claim that the set of k-edge connected components of a graph is unique.

Lemma 3.1: *Given a graph G, all its k-edge connected components are unique.* □

Proof Sketch: We prove it by contradiction. Assume that there are two different sets of k-edge connected components. Then there must exist two k-edge connected components from each one of the two sets, g_1 and g_2, that are non-disjoint and also non-identical. Otherwise, the two sets of k-edge connected components are the same. Therefore, $(g_1 \cup g_2)$ is also k-connected (since there is no cut of value less than k that can disconnect $(g_1 \cup g_2)$), which contradicts that both g_1 and g_2 are k-edge connected components which should be maximal. □

3.1 Graph Decomposition-based Framework

Given a graph G and an integer k, our graph decomposition-based framework for computing k-edge connected components is iteratively decomposing a non k-connected subgraph into several (possibly more than two) connected subgraphs by removing edges in all cuts of G with values less than k. More specifically, initially, we have one connected graph which is G, and whenever there are connected subgraphs that are not k-connected, we try to decompose them into sets of smaller subgraphs. The algorithm terminates when all connected subgraphs are k-connected. The pseudocode of our framework is shown in Alg. 1, where Decompose denotes an algorithm to decompose a graph which will be discussed shortly.

The connected subgraphs (including the intermediate subgraphs

Algorithm 1 Computing k-edge connected components

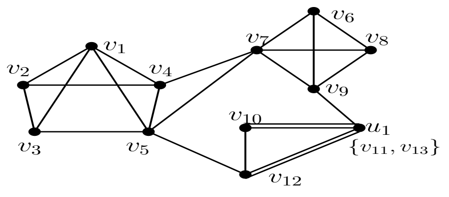
```
Input: A graph $G = (V, E)$ and an integer $k$.
Output: $k$-edge connected components of $G$.

1: Initialize a queue $Q_g$ consisting of a single graph $G$;
2: for each subgraph $g$ in $Q_g$ do
3:     $\phi_k(g) \leftarrow$ Decompose$(g, k)$;
4:     if $\phi_k(g)$ consists of only one subgraph then
5:         Output $\phi_k(g)$ as one of the $k$-edge connected components;
6:     else
7:         Push all subgraphs of $\phi_k(g)$ into $Q_g$;
```

and the final k-edge connected components) can be organized into a tree structure as depicted in Fig. 2, called *decomposition tree*, where each ellipse represents a connected subgraph. The root represents the input graph G, and the children of a connected subgraph represent the set of connected subgraphs obtained by decomposing that subgraph (using Decompose). Therefore, all k-edge connected components of G are represented by leaf nodes of the tree. Decomposition tree is further characterized in Section 3.3. Alg. 1 generates all k-edge connected components by conducting a Breadth-First Search traversal on the decomposition tree.

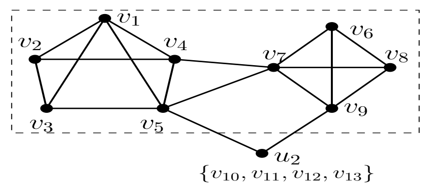
We define a *partition graph PG* and two operators, the *merge operator* $\rho_{u,v}(PG)$ and the *split operator* $\gamma_{S,T}(PG)$, on it in the following.

Partition Graph: A partition graph $PG = (G, D)$ of a graph $G = (V, E)$ is obtained from G by augmenting each vertex $u \in V$ with a set of elements from domain D, such that $\cup_{u \in V} D(u) = D$ and $D(u) \cap D(v) = \emptyset, \forall u \neq v$, where $D(u)$ denotes the set of elements associated with vertex $u \in V$. That is, sets of elements associated with the vertices in V form a partition of D.

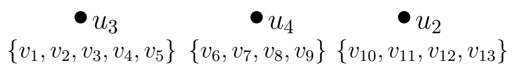
An ordinary graph $G = (V, E)$ is a special partition graph with $D = V$, where $D(u) = \{u\}, \forall u \in V$. Therefore, the input to our algorithm is a partition graph $PG = (G = (V, E), D(= V))$ where $D(u) = \{u\}, \forall u \in V$.



(a) Result of merging v_{11}, v_{13}



(b) Result of merging v_{10}, v_{12}, u_1


u_3
u_4
u_2
$\{v_1, v_2, v_3, v_4, v_5\}$
$\{v_6, v_7, v_8, v_9\}$
$\{v_{10}, v_{11}, v_{12}, v_{13}\}$

(c) Final partition graph

Figure 4: Partition graphs

Merge Operator: A merge operator $\rho_{u,v}(PG)$ merges two vertices u and v of PG into a single super-vertex, denote the new super-vertex by v_{uv}. More specifically, the operator adds v_{uv} to PG with $D(v_{uv}) = D(u) \cup D(v)$, and adds parallel edges (v_{uv}, x) to PG for each $x \in V \backslash \{u, v\}$, where the number of parallel edges is equal to the total number of parallel edges (u, x) and (v, x), and then removes vertices u and v and their associated edges from PG. Note that, after the merge, edge (u, v) is removed if it is in PG, and other edges are retained.

For example, Fig. 4(a) shows the result of the merge operator $\rho_{v_{11},v_{13}}(PG)$ where PG is the partition graph implied by Fig. 3. There are two parallel edges between v_{10} and u_1. Edge (v_{11}, v_{13}) is removed in the resulting partition graph.

Split Operator: A split operator $\gamma_{S,T}(PG)$ removes all edges of cut C from PG, where $C = (S, T)$ is a cut of PG.

Given an arbitrary partition graph $PG = (G = (V, E), D)$, the resulting graph obtained by performing either the merge operator or the split operator is still a partition graph. To simplify the presentation, in the rest of the paper, we use $\rho_{s,t}(PG)$ and $\gamma_{S,T}(PG)$ to denote the resulting partition graphs after applying the corresponding operators, respectively.

Graph Decomposition: The general idea of our graph decomposition algorithm, Decompose, is iteratively applying merge and split operators on the partition graph $PG = (G = (V, E), D(= V))$ to finally obtain a special partition graph $PG_r = (G_r = (V_r, E_r), D(= V))$ with $E_r = \emptyset$, such that each $D(u), u \in V_r$, induces a connected subgraph of G. Let $\phi_k(PG_r)$ denote the set of connected subgraphs represented by PG_r, i.e. $\phi_k(PG_r) = \{G[D(u)] \mid u \in V_r\}$. Decompose returns $\phi_k(PG_r)$ as the decomposed graph. For example, for the graph in Fig. 3, if we decompose it for $k = 3$, the final special partition graph will be in the form of Fig. 4(c), from which we can see that $\phi_3(G) = \{G_1, G_2, G_3\}$. The pseudocode of Decompose is shown in Alg. 2. Given an input graph G and an integer k. The algorithm first constructs the corresponding partition graph PG (Line 1). Then, it proceeds iteratively to update the partition graph until its edge set is empty (Lines 3-8), through the split operator (Line 5) and the merge operator (Line 7). To choose which operator to be applied depends on the value of the cut found by MinCut (Line 3) which computes the minimum cut for a pair of vertices.

Algorithm 2 Decompose(G, k)

```
1: Construct the corresponding partition graph $PG$ of $G$, $PG_0 \leftarrow ((G_0 \leftarrow G), (D \leftarrow V))$, $i \leftarrow 0$;
2: while The edge set of $PG_i$ is non-empty do
3:     $(s, t, S, T) \leftarrow$ MinCut($PG_i, k$);
4:     if $w(S, T) < k$ then
5:         $PG_{i+1} \leftarrow \gamma_{S,T}(PG_i)$;
6:     else
7:         $PG_{i+1} \leftarrow \rho_{s,t}(PG_i)$;
8:     $i \leftarrow i + 1$;
9: return $\phi_k(PG_i)$;
```

Example 3.1: Consider the graph in Fig. 3 and $k = 3$. Assume the first iteration of computing minimum cut is computed between vertices v_{11} and v_{13}. As the value of the cut is 3, v_{11} and v_{13} are merged and the resulting partition graph is shown in Fig. 4(a). Fig. 4(b) shows the partition graph after merging v_{10}, v_{12}, u_1. The next pair of vertices, whose minimum cut is computed, is v_9 and u_2, then edges in the cut $C = \{(v_5, u_2), (v_9, u_2)\}$ are removed since the cut consists of only two edges. The resulting partition graph is the union of the subgraph shown in the dotted rectangle in Fig. 4(b) and the isolated vertex u_2. After applying a few more merge operators and split operators, we will get the special partition graph shown in Fig. 4(c) from which we can get the 3-edge connected components. □

3.2 Algorithm Correctness

In this section, our goal is to prove the correctness of the framework (Alg. 1) provided that MinCut in Alg. 2 correctly returns the minimum cut for a pair of vertices.

Lemma 3.2: *Given a partition graph PG, after applying a total number of* $(|V| - 1)$ *of merge operators and split operators,* Decompose *will produce a partition graph with no edges.* □

Proof Sketch: We prove it by induction. (Base Case) for $|V| = 1$, the claim is trivially true. (General Case) for $|V| > 1$, we consider two cases based on the type of the first operator applied. If the first operator is a merge operator, then the resulting partition graph has $|V'| = |V| - 1$ vertices which requires $(|V'| - 1)$ iterations of merge operators and split operators to remove all the edges. Therefore, the claim is true. If the first operator is a split operator, assuming that the split operator divides the partition graph into two disconnected subgraphs with $|V_1|$ and $|V_2|$ vertices each. Then all edges can be removed in $1 + (|V_1| - 1) + (|V_2| - 1)$ operations. Therefore, the claim is true. □

Atomicity property: Here, we prove the atomicity of Decompose, i.e., each of the connected subgraphs returned by Decompose will contain either all vertices of a k-edge connected component or none of its vertices.

Proof Sketch: To prove the claim, we prove that all edges in a k-edge connected component $g = (V_g, E_g)$ will not be removed during the execution of Decompose. As the algorithm removes only edges in cuts with values less than k, it is equivalent to prove that any edge of g will not be included in any cut with value less than k.

We prove it by contradiction. Assume that some of the edges of g are included in cuts with values less than k, and let (u, v) be the first such edge included in a cut C; that is, cuts are considered in the order of being found by iterations of MinCut. Let V_1 and V_2 denote the vertices of the two disconnected subgraphs that contain u and v respectively after removing all edges in C. Then $C \cap E_g$ is a cut of g which cuts V_g into $V_1 \cap V_g$ and $V_2 \cap V_g$. This means that there is a cut $C \cap E_g$ of g with value less than k, which contradicts that g is a k-edge connected component. □

Cutability property: To prove the cutability of Decompose, we prove that if a graph is not k-connected, then Decompose will decompose it into at least two disconnected subgraphs.

Proof Sketch: Given a non k-connected graph G, there must exist cuts of value less than k that can cut G into at least two disconnected subgraphs. The rest is to prove that Decompose can find at least one such cut eventually during the decomposition process.

We prove it by contradiction. Assume that Decompose are not able to find any cut of value less than k during the decomposition process. Consider such a cut C of value less than k. For any edge $(u, v) \in C$, u and v are not going to be merged since they are not k-connected. Therefore, all the edges in C are retained after merge operators, which contradicts that there is no edge remaining in the final partition graph obtained by Decompose (Lemma 3.2). □

Having shown the atomicity property and the cutability property of Decompose, we are now ready to show the correctness of the graph decomposition-based technique for computing k-edge connected components (Alg. 1) of a graph.

Theorem 3.1: *Alg. 1 correctly computes all k-edge connected components of a graph, provided that* MinCut *in Alg. 2 returns the minimum cut for a pair of vertices.* □

Proof Sketch: From the atomicity property, we know that a k-edge connected component will never be further decomposed into multiple subgraphs during the decomposition process (atomicity property). As long as a graph is not k-connected, Decompose will decompose it into at least two disconnected subgraphs (cutability property). Therefore, the input graph will be decomposed into a set of k-edge connected components by Alg. 1 eventually. □

3.3 Complexity

Time complexity of Alg. 1: Let h denote the height of the decomposition tree as illustrated in Fig. 2 and defined in Section 3.1, and $T_{\mathsf{Dec}}(G)$ denote the time complexity of decomposing a graph G using Decompose, which will be discussed in the next section. Then, the time complexity of Alg. 1 is $O(h \times T_{\mathsf{Dec}}(G))$.

Characterization of the decomposition tree: A decomposition tree of G, as depicted in Fig. 2, is a representation of the iterative process that decomposes non k-connected subgraphs of G into smaller subgraphs. Each k-edge connected component of G is represented by a leaf node, and each intermediate node is a connected subgraph of G that is not k-connected itself. The sibling nodes (subgraphs), e.g., $G_{1,1}, \ldots, G_{1,m}$, represent the result of decomposing its parent graph, G_1.

In a simple way, the sibling subgraphs have the property that each of them is connected to the remaining part of its parent graph by less than k edges except one sibling subgraph. For example, consider the graph in Fig. 1 and $k = 3$, G_1, G_2, G_3 form the sibling subgraphs for the parent graph G. Both G_1 and G_3 are connected to the remaining part of G by 2 edges, while G_2 is connected to the remaining part of G by 4 edges. Now, consider another graph in Fig. 3 and $k = 3$, $(G_1 \cup G_2), G_3$ form the sibling subgraphs, because both G_1 and G_2 are connected to the remaining part of the parent graph by 3 edges. More rigorously, the sibling subgraphs have the property that each of them is connected to any vertex in the remaining part of the parent graph by less than k edge-disjoint paths. Given a graph $G = (V, E)$, we define a partition of V based on the connectivity between pairs of vertices, denote by $\mathcal{V}_p = \{V_1, \cdots, V_m\}$, such that each pair of vertices in the same partition is k-connected in G, and otherwise not k-connected. Then, the vertex set of each subgraph obtained by Decompose(G, k) is contained in exactly one of the partitions of $\mathcal{V}_p$. As a result, a non k-connected graph G will be decomposed into at least $m \geq 2$ disconnected subgraphs. For example, if we attach a four-vertex clique G_4 to be adjacent to v_{16} in Fig. 1 by two edges, then Decompose can decompose the graph into four subgraphs G_1, G_2, G_3, and G_4, in one iteration.

By iteratively constructing children subgraphs for each non k-connected subgraph in the above way as defining the sibling subgraphs, the decomposition tree can be constructed. Each level of the tree can be computed from the union of their parent graphs by a single iteration of Decompose. The height h of the decomposition tree reflects the "depth" of the k-edge connected components nested together to form G. After each iteration of Decompose, the "depth" of nestedness is reduced by one. For example, in Fig. 1, the "depth" is two. After the first iteration of Decompose with $k = 3$, there are three disconnected subgraphs G_1, G_2, and G_3, where G_2 consists of two k-edge connected components nested together and can now be decomposed in the next iteration.

Potentially a smaller h: Intuitively, given a non k-connected graph G, the more the number of disconnected subgraphs returned by Decompose, the shorter the height h of the decomposition tree. The minimum number of disconnected subgraphs that will be returned by Decompose is guaranteed by the number of sibling subgraphs as discussed above. Usually, we will get more disconnected subgraphs. For example, consider the graph in Fig. 3, there are only two sibling subgraphs $(G_1 \cup G_2)$ and G_3 as discussed above. However, in practice, if G_3 is split from G before any merge of a pair of vertices where one is from G_1 and the other is from G_2, then G_1 and G_2 will also be split by a following split operator in the same iteration of Decompose. Therefore, three disconnected subgraphs will be returned by Decompose, and we get $h = 1$. This phenomenon is observed in our experimental studies. For example, each of the Amazon and web-Google dataset has almost 1,000 10-edge connected components, however, h is only 3. We also observe that, in our testing, the small value of h is quite steady across different sizes of graphs and different numbers of k-edge connected components in a graph.

4. GRAPH DECOMPOSITION

In this section, we first discuss a baseline algorithm for graph decomposition, and then propose a simple yet efficient data structure to replace the Fibonacci heap as required by the baseline algorithm. Finally, we propose several optimization techniques to further improve the performance of our decomposition approach based on the threshold k of computing k-edge connected components.

4.1 Baseline algorithm

The baseline algorithm for graph decomposition is by calling the procedure of maximum adjacency search (Mas in Section 2.2) to compute the minimum cut between a pair of vertices, i.e., replace MinCut in Alg. 2 by Mas, denoted by BaseLine. Recall that, Mas computes a linear order L for the vertices of G, and let s and t be the two vertices that are most recently added to L, and t be the last one, then the adjacent edges of t is a minimum s–t cut.

Iterations	List L
1	$v_1, v_2, v_3, v_4, v_5, v_7, v_6, v_8, v_9, v_{11}, v_{12}, v_{10}, v_{13}$
2	$v_1, v_2, v_3, v_4, v_5, v_7, v_6, v_8, v_9, v_{11}, v_{12}, \{v_{10}, v_{13}\}$
3	$v_1, v_2, v_3, v_4, v_5, v_7, v_6, v_8, v_9, v_{11}, \{v_{10}, v_{12}, v_{13}\}$
4	$v_1, v_2, v_3, v_4, v_5, v_7, v_6, v_8, v_9, \{v_{10}, v_{11}, v_{12}, v_{13}\}$
5	$v_1, v_2, v_3, v_4, v_5, v_7, v_6, v_8, v_9$
6	$v_1, v_2, v_3, v_4, v_5, v_7, \{v_8, v_9\}, v_6$
7	$v_1, v_2, v_3, v_4, v_5, v_7, \{v_6, v_8, v_9\}$
8	$v_1, v_2, v_3, v_4, v_5, \{v_6, v_7, v_8, v_9\}$
9	v_1, v_2, v_3, v_4, v_5
10	$v_1, \{v_4, v_5\}, v_2, v_3$
11	$v_1, \{v_2, v_3\}, \{v_4, v_5\}$
12	$v_1, \{v_2, v_3, v_4, v_5\}$

Table 1: Execution of BaseLine

Example 4.1: Consider the graph in Fig. 3 and $k = 3$. The first iteration of Mas will produce the order $L = (v_1, v_2, v_3, v_4, v_5, v_7, v_6, v_8, v_9, v_{11}, v_{12}, v_{10}, v_{13})$. Then $s = v_{10}$ and $t = v_{13}$, and it can be verified that $\{(v_{10}, v_{13}), (v_{11}, v_{13}), (v_{12}, v_{13})\}$ is a minimum v_{10}–v_{13} cut.

The lists L obtained from all 12 iterations of Mas during the running of BaseLine are shown in Table 1, where super-vertices are denoted by their associated elements. We omit those isolated vertices. For the first three iterations, we apply the merge operator since the value of cut found is no less than 3, and the resulting partition graph after the first three iterations is shown in Fig. 4(b). After the fourth iteration, we find a cut with value 2, then the split operator is applied. The graph is decomposed into three disconnected subgraphs after 12 iterations. □

Time complexity of BaseLine: Since the time complexity of Mas is $O(|E| + |V| \log |V|)$ (Theorem 2.1), and BaseLine will invoke at most $(|V| - 1)$ iterations of Mas (Lemma 3.2), the time complexity of BaseLine thus is $O(|V||E| + |V|^2 \log |V|)$.

4.2 A Simple Efficient Data Structure

The term $|V| \log |V|$ in the time complexity of BaseLine is due to finding the most tightly connected vertex in Mas using the Fibonacci heap, which assumes that the *update-key* operation and the *extract-max* operation takes $O(1)$ and $O(\log |V|)$ time, respectively. Note that extract-max and update-key are the only two operations of Fibonacci heap needed in Mas, assuming that *insert* and *remove* are special cases of the update-key operation. However, the Fibonacci heap is complicated and does not guarantee good performance in practice for a reasonable size graph. Motivated by this, we propose a simple yet efficient data structure to accelerate BaseLine as follows.

Data Structure: Let $key(v)$ denote the key of vertex v during the execution of Mas, i.e., $key(v) = w(L, v)$ which is the number of edges between v and vertices in L. Let $\bar{c}$ denote the maximum value among all minimum cuts for different pairs of vertices. We have $key(v) \le \bar{c}, \forall v \in V$, and $\bar{c} \le |E|$. We propose to use doubly linked lists coupled with a head table to accommodate the *extract-max* and *update-key* operations.

For the doubly linked lists, we use an array of size $3|V|$ to store the data; that is, for each vertex $u \in V$, $key(u)$ stores the key, $pre(u)$ and $next(u)$ store the predecessor and successor vertices in the doubly linked list, respectively. Each doubly linked list links all the vertices with identical keys. In the head table H, $H(x)$ stores the first vertex in the doubly linked list whose key value equals to x. For example, in Fig. 5, assume that vertices v_i and v_j have the same key value x, and v_i is the first vertex in this doubly linked list, then $H(x) = v_i$, $pre(v_j) = v_i$, and $next(v_i) = v_j$. Since the maximum key value can be potentially be $\bar{c}$, the size of head table H is $\bar{c}$ in worst case. We discuss in the next subsection how to lower the size of H.

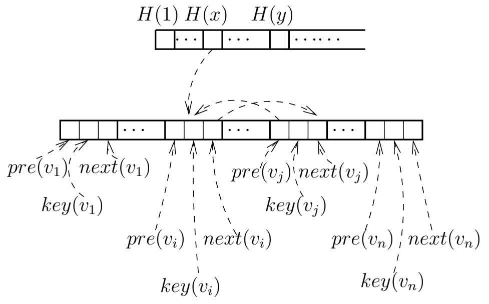


Figure 5: Data Structure

In addition to the two arrays, we also record a value po, which is the potential of the current maximum key value for all vertices in the data structure, and is initialized as 0. The value of po is updated after only the update-key operation.

Extract-max: To extract the next vertex with the largest key value, we first decrease po until $H(po) \ne nil$, and then report the first vertex pointed by $H(po)$ and remove it from that doubly linked list, as shown below.

```
Extract-max($u$):
  while $H(po) = nil$ do
      $po \leftarrow po - 1$;
  $u \leftarrow H(po)$, and remove $u$ from $H(po)$;
```

Update-key: To update the key of vertex v from x to y, we first remove v from the doubly linked list represented by $H(x)$, and then insert v to the doubly linked list $H(y)$. We also update $po \leftarrow y$ if $y > po$.

```
Update-key($v$):
  remove $v$ from the doubly linked list $H(key(v))$;
  update $key(v)$ and insert $v$ into $H(key(v))$;
  $po \leftarrow key(v)$ if $key(v) > po$;
```

Example 4.2: Consider Fig. 5, assume that $H(x)$ denotes a doubly linked list containing vertices v_i and v_j, i.e., $key(v_i) = key(v_j) = x$, and v_i is the first vertex in the doubly linked list. Assume po is now set as y and the largest key value with non-empty doubly linked list is x. To extract the vertex with maximum key value, we first decrease po to x which takes $O(y - x)$ time, and then report and remove v_i which takes $O(1)$ time. □

Time Complexity: The worst time complexity of the update-key operation is $O(1)$ as ensured by the properties of doubly linked lists. In the following, we discuss the time complexity of the extract-max operation which is nontrivial.

Lemma 4.1: *The worst time complexity of extract-max is $O(\bar{c})$, which can be as large as $|E|$.* □

Proof Sketch: For extract-max, we need to decrease po until $H(po) \ne nil$. As po can be potentially as large as $\bar{c}$ and the only non-empty doubly linked list could be $H(1)$, therefore, in worst case, extract-max takes $O(\bar{c})$ time. □

Although the worst time complexity of one extract-max is bounded by $|E|$, we show that the total running time of $|V|$ extract-max operations is still bounded by $|E|$ through the following theorem.

Theorem 4.1: *Given a graph $G = (V, E)$,* Mas *can find a minimum cut for an arbitrary vertex pair s and t in time $O(|E|)$ using our data structure, denoted by* Mas-Linear*.* □

Proof Sketch: The correctness of Mas-Linear directly follows from the correctness of extract-max and update-key using our data structure and the correctness of Mas.

Let $In(i)$ and $De(i)$ denote the increase value and decrease value of po when finding the most tightly connected vertex for the i-th time in Mas, i.e., to compute the i-th vertex of L, respectively. Then, the time complexity is bounded by $O(|E|+\sum_{j=1}^{|V|} De(j))$, where $O(|E|)$ is the total time of update-key operations and $\sum_{j=1}^{|V|} De(j)$ is the total time of extract-max operations. We have the property that $\sum_{j=1}^{|V|} In(j) \le |E|$ since the update-key operation is executed once for each edge in the graph and the value of po is increased by the amount of at most one after each update-key operation. Then, $\sum_{j=1}^{|V|} De(j) \le \sum_{j=1}^{|V|} In(j) \le |E|$. Therefore, the total time complexity of Mas-Linear follows. □

4.3 Optimization

In the following, we propose three optimization techniques to further improve the performance of Decompose based on the threshold k of computing k-edge connected components.

Early merge based on k-connectivity. From Line 7 of Alg. 2, we can see that the only requirement for applying the merge operator is that the connectivity between s and t should be no less than k. Therefore, we can merge more than one pair of vertices during one iteration of Mas, as long as each pair of vertices is guaranteed to be k-connected in the input graph, based on the following lemmas. Recall that, after finding a minimum cut, Alg. 2 merges only one pair of vertices if applicable.

Consider the list L obtained by Mas on graph G. For any $v \in L$, let L_v denote the list of vertices added to L prior to v (excluding v), and p_v denote the last vertex in L_v. We call $w(L_v, v)$ as the key value of $v, \forall v \in V$. Then, we have the following property.

Lemma 4.2: *Given a graph G, for any vertex $v \in L$, if $w(L_v, v) \ge k$, then p_v and v are k-connected in G.* □

Proof Sketch: Consider the subgraph of G induced by the vertex set $L_v \cup \{v\}$, i.e., $G[L_v \cup \{v\}]$, applying Mas on it will produce exactly the same list $L_v \cup \{v\}$. This is because that, considering the two executions of Mas on G and $G[L_v \cup \{v\}]$ respectively and assuming that the same initial vertex (i.e., the first vertex in L_v) is chosen, then, for any vertex $u \in L_v \cup \{v\}$, its key value will be the same for both executions. Therefore, provided that $w(L_v, v) \ge k$, p_v and v are k-connected in $G[L_v \cup \{v\}]$ (Theorem 2.1), which implies that p_v and v are k-connected in G. □

Following Lemma 4.2, we can merge more than one pair of vertices after obtaining the list L. However, this will result in one additional scan on L of size $|V|$. Therefore, we propose to merge pairs of vertices on-the-fly following the lemma below.

Lemma 4.3: *During the execution of* Mas*, whenever there is a vertex v with key value no less than k, we then apply the merge operator on u and v where u is the vertex last added to the current list L, and continue this execution.* □

Proof Sketch: Let L denote the list obtained by Mas, and L_1 denote the list obtained by merging all pairs of vertices of L satisfying Lemma 4.2. Note that, in L_1, it merges only pairs of consecutive vertices in L. Since L_1 is obtained by getting L first, the set of merge operators will not affect the correctness of the algorithm as guaranteed by Lemma 4.2. Let L_2 denote the list obtained by this lemma. We prove that L_2 is exactly the same as L_1.

Note that, we consider each super-vertex , after merging pairs of vertices, in L_1 and L_2 as a set of vertices, and call it an item of L_1 and L_2. Let $L_{1,i}$ and $L_{2,i}$ denote the i-th item of L_1 and L_2, respectively. We prove that $L_1 = L_2$ by induction. For $i = 1$, if $L_{1,1}$ consists of only one vertex u, which means that $w(u, v) < k, \forall v \neq u$, then $L_{1,1} = L_{2,1}$. Otherwise, for any vertex v other than the initial vertex u in $L_{1,1}$, we have $w(L_v, v) \geq k$, then v will be included in $L_{2,1}$; for any vertex $v \notin L_{1,1}$, we have $w(L_v, v) < k$, then v will not be included in $L_{2,1}$. Therefore $L_{1,1} = L_{2,1}$. For $i > 1$, the first vertex in $L_{1,i}$ and $L_{2,i}$ is identical as guaranteed by the selection of vertex with maximum key value. Similarly to the proof for the case $i = 1$, we have $L_{1,i} = L_{2,i}$. Therefore, $L_1 = L_2$. □

Following Lemma 4.3, consider the graph in Fig. 3 with $k = 3$, when L is initialized as $\{v_1\}$, the list L obtained after the first execution of Mas is $(v_1, v_2, v_3, \{v_4, v_5\}, v_7, v_6, \{v_8, v_9\}, v_{11}, v_{12}, \{v_{10}, v_{13}\})$, in which three pairs of vertices are merged in one iteration.

List sharing after applying split operators. Consider Alg. 2, if a minimum cut of value less than k is found by Mas, after applying the split operator, a minimum cut of the resulting graph can be determined from the same list L, without actually running Mas on the resulting graph. Therefore, the list L can be shared across different executions of Mas after applying split operators.

Lemma 4.4: *Given a graph G, if the value of a minimum s–t cut (S, T) returned by* Mas *is less than k, then, in the resulting graph after applying the split operator, i.e., $\gamma_{S,T}(G)$, $(S\backslash\{s\}, \{s\})$ is a minimum r–s cut, where r is the vertex added to L prior to s.* □

Proof Sketch: Note that, in the cut (S, T), T consists of only a single vertex t. After the split operator which removes all adjacent edges of t from G, we obtain $\gamma_{S,T}(G)$. The list L' obtained by applying Mas on $\gamma_{S,T}(G)$ will be exactly the same as S. Therefore, $(S\backslash\{s\}, \{s\})$ is a minimum r–s cut in $\gamma_{S,T}(G)$. □

Consider the execution example shown in Table 1, the minimum cut obtained at iteration 4 has value 2 which is less than $k = 3$. Therefore, we can get the list L of iteration 5 without running Mas again. Note that, this optimization can be applied iteratively until a merge operator is applied.

Efficient vertex reduction by degree. In a k-edge connected component, each vertex will have degree at least k. Therefore, we can remove all vertices with degrees less than k from the input graph. Efficient algorithm to recursively remove all vertices with degree less than k is studied in [2], which runs in $O(|V| + |E|)$ time. The general idea is that, it maintains a queue Q of vertices which is initialized to contain all vertices with degree less than k in the current graph G. It iteratively removes the vertices in Q from G. After the removal of vertices, the degrees of some vertices not in Q will change from at least k to below k, and then those vertices are added to Q and the process of removing vertices from G continues until no new vertex is added to Q. We use this efficient vertex reduction method based on degree k in our optimization. For more details, please refer to [2]. Without this optimization, to remove vertices with degrees less than k would require scanning the graph multiple times.

4.4 Algorithm with Optimization

The overall algorithm by incorporating the optimization techniques is shown in Alg. 3, denoted by Decompose-LMS, where the **L**inear data structure, early **M**erge, and list **S**haring optimization techniques are used. In procedure Mas-LMS, the vertices in queue Q are recursively merged with u (Lines 13-20), and the split operator is recursively applied if the minimum cut implied by the current list L has value less than k (Lines 21-23).

Algorithm 3 Decompose-LMS (G, k)

```
Input: A graph $G = (V, E)$ and an integer $k$.
Output: Subgraphs of $G$ if $\lambda(G) < k$, and $G$ otherwise.

 1: Construct the corresponding partition graph $PG$ of $G$, $PG_0 \leftarrow (G_0(\leftarrow G), D(\leftarrow V))$, $i \leftarrow 0$;
 2: while The edge set of $PG_i$ is non-empty do
 3:     $PG_{i+1} \leftarrow$ Mas-LMS $(PG_i, k)$;
 4:     $i \leftarrow i + 1$;
 5: return $\phi_k(PG_i)$;

 6: procedure Mas-LMS $(G, k)$
 7: $L \leftarrow$ {an arbitrary vertex $u$ of $V$};
 8: Initialize our data structure;
 9: while $L \neq V$ do
10:     $u \leftarrow$ extract-max;
11:     Add $u$ to $L$ and remove $u$ from the data structure;
12:     Initialize a queue $Q$ with $u$;
13:     while $Q \neq \emptyset$ do
14:         $v \leftarrow Q.pop()$;
15:         for each $(v, s) \in E$ with $s \notin L$ do
16:             if the key of $s$ increases to pass $k$ then
17:                 Add $s$ to $Q$, remove $s$ from the data structure;
18:             else
19:                 Update-key for $s$;
20:         Merge $u$ and $v$ if $u \neq v$;
21: while $|L| > 1$ and the value of the cut implied by the last two vertices in $L$ is less than $k$ do
22:     Split the cut;
23:     Remove the last vertex from $L$;
```

In Decompose-LMS where the optimization of early merge based on k-connectivity is applied, whenever the key of a vertex v is increased to pass k, v is removed from our data structure. Therefore, the maximum key value in our data structure is bounded by k, which means that we need only k entries for our head table which is discussed in Section 4.2.

Theorem 4.2: *Given a graph $G = (V, E)$,* Decompose-LMS *correctly decomposes it into at least two disconnected subgraphs if $\lambda(G) < k$ and returns G otherwise. The time complexity is $O(l\times|E|)$, where l $(\leq |V|)$ is the number of iterations of* Mas-LMS*, i.e., the value of i at Line 5 of Alg. 3.* □

Proof Sketch: The correctness of Decompose-LMS directly follows from the correctness of Mas, and Lemmas 4.2, 4.3, and 4.4. For the time complexity, since Mas-LMS takes $O(|E|)$ time and there are totally l iterations of Mas-LMS, the time complexity of Decompose-LMS follows. □

Bound on the value of l: Now, we characterize the value of l in two situations based on whether G is k-connected or not.

Given a non k-connected graph $G = (V, E)$, let $\mathcal{G}_k = \{g_1 = (V_1, E_1), \cdots, g_m = (V_m, E_m)\}$ be the set of k-edge connected components of G. Then, $l \leq \max_{j=1}^{m} |V_j|$. The reason is that, for any two k-edge connected components of G, g_i and g_j, Decompose-

LMS either splits the two subgraphs or merges them into a single subgraph. If g_i and g_j are split into two subgraphs by Decompose-LMS, then the number of vertices of g_i and g_j is each reduced by at least one after one iteration of Mas-LMS. The reason is that, let u_i and u_j be the last node of g_i and g_j on L respectively, then the keys of u_i and u_j are at least k, therefore, u_i and u_j are merged with other vertices. Otherwise, vertices of g_i and g_j are merged into super-vertices. Let sg_i^a and sg_j^a denote the subgraph induced by vertices (or super-vertices) of g_i and g_j after a iterations of Mas-LMS, respectively. Note that super-vertices in sg_i^a and sg_j^a can overlap. Then the number of vertices in sg_i^{a+1} (sg_j^{a+1}) is strictly less than sg_i^a (sg_j^a), since both sg_i^a and sg_j^a are k-connected. Therefore, after each iteration of Mas-LMS, the number of vertices in each of g_i will be reduced by at least one, and $l \leq \max_{j=1}^{m} |V_j|$.

Given a k-connected graph G, each vertex of G will have a degree at least k and Decompose-LMS will merge all vertices in G into a single super-vertex. In Decompose-LMS, an iteration of Mas-LMS computes an order L of vertices in G and merges each such vertex having $w(L_v, v) \geq k$ with its predecessor vertex in L (refer to early merge optimization). Let $k(L)$ denote the number of vertices having $w(L_v, v) \geq k$ in L. Then, after each iteration of Mas-LMS which computes an order L, the number of vertices is reduced by $k(L)$. Recall that $w(L_v, v)$ is the number of edges between v and vertices prior to v in L. Implicitly, for a pair of vertices u and v that have more than $k-1$ parallel edges between them, Mas-LMS ensures to merge the two vertices into a super-vertex. Given a random order L and a vertex v with degree no less than $2k$, the probability that v will be merged with other vertices is at least $\frac{1}{2}$. Let V_{2k} denote the subset of vertices in G whose degrees are no less than $2k$. The expected number of vertices that will be reduced after one iteration of Mas-LMS is at least $\frac{|V_{2k}|}{2}$. After merging vertices into super-vertices, degrees of super-vertices are tend to become larger. Therefore, the number of vertices reduces rapidly after each iteration of Mas-LMS. For a special graph that every vertex has degree no less than $2k$, the number of iterations l is expected to be bounded by $\log |V|$. The same phenomenon also applies to each k-edge connected component of G if G is not k-connected.

In our empirical studies, the sizes of the largest k-edge connected components of as-Skitter and SSCA-20 dataset are over 0.3 million and 1 million, while the value of l is 9 and 8, respectively. Of all the real and synthetic graphs we tested, the values of l are no more than 21 regardless of the sizes of graphs and the sizes and/or numbers of k-edge connected components of a graph. Therefore, algorithm Decompose-LMS is able to process large graphs. Note that our optimization techniques are inapplicable to those connectivity testing algorithms (such as TestConnect [11] as discussed in Section 2.2) which are based on maximum flow techniques, since flows computed for different pairs of vertices are unrelated.

Example 4.3: Given the graph in Fig. 3 and $k = 3$. The list L obtained for each iteration of Mas-LMS is shown in Table 2, where super-vertices are denoted by their associated elements. Decompose-LMS computes a graph decomposition for $k = 3$ in four iterations. Therefore, Decompose-LMS is much faster than BaseLine (Section 4.1), which takes 12 iterations as shown in Table 1. □

Iterations	List L
1	$v_1, v_2, v_3, \{v_4, v_5\}, v_7, v_6, \{v_8, v_9\}, v_{11}, v_{12}, \{v_{10}, v_{13}\}$
2	$v_1, \{v_4, v_5\}, \{v_2, v_3\}, v_7, \{v_6, v_8, v_9\}, v_{11}, \{v_{10}, v_{12}, v_{13}\}$
3	$v_1, \{v_2, v_3, v_4, v_5\}, \{v_6, v_7, v_8, v_9\}, \{v_{10}, v_{11}, v_{12}, v_{13}\}$
4	$\{v_1, v_2, v_3, v_4, v_5\}$

Table 2: Execution of Decompose-LMS

Remarks on the range of the value of h: Our algorithm Decompose-LMS will decompose a graph G into a set of subgraphs sg_i, each of which possibly contains several k-edge connected components. Assume that two different k-edge connected components g_i and g_j have equal probabilities to be split and to be merged. Then, the decomposition tree (see Section 3) tends to be a balanced tree on average, and h is expected to be bounded by $\log |V|$. In our empirical studies, the largest h is 5 across graphs whose vertex numbers vary from 4 thousand to 2 million.

5. EXPERIMENTS

We conduct extensive performance studies to evaluate the efficiency of our framework (proposed in Section 3) and our graph decomposition algorithm based on several optimization techniques (studied in Section 4) for the computation of k-edge connected components in graphs. The following algorithms are implemented:

- CutB-GMC: The cut-based framework plugged in a variant of finding **G**lobal **M**in-**C**ut algorithm as discussed in Section 2.2. We also incorporate the optimization techniques and pruning rules proposed in [20] into CutB-GMC.
- CutB-TC: The cut-based framework plugged in the **T**esting **C**onnectivity algorithm (TestConnect) which can find a minimum cut with value less than k by a slight modification, as discussed in Section 2.2.[1]
- DecB-LMSD: Our graph decomposition-based framework (Section 3) plugged in our graph decomposition algorithm with all the optimization techniques proposed in Section 4, i.e., the framework in Alg. 1 plus the decomposition algorithm in Alg. 3.
- DecB: Our graph decomposition-based framework plugged in the baseline algorithm BaseLine for graph decomposition.
- CutB-LMD: The cut-based framework plugged in a modification of our decomposition algorithm to cut a graph into only two disconnected subgraphs. All the optimization techniques proposed in Section 4 are applied except the list sharing optimization which is inapplicable.

All algorithms are implemented in C++ and compiled with GNU GCC with the -O3 optimization. All experiments are conducted on a PC with an Intel(R) Core(TM) i5-2400 CPU (3.10GHz) and 4GB memory running Ubuntu 12.04. We evaluate the performance of all algorithms on both real and synthetic graphs as follows.

Real Graphs: We evaluate the algorithms on eight real graphs, Arxiv General Relativity collaboration network (ca-GrQc), Arxiv Condensed Matter collaboration network (ca-CondMat), email network from a EU research institution (email-EuAll), who-trusts-whom network of Epinions.com (soc-Epinions1), Amazon product co-purchasing network (amazon0601), web graph from Google (web-Google), Wikipedia talk (communication) network (wiki-Talk), and Internet topology graph (as-Skitter). All the graphs are downloaded from the Stanford SNAP library[2], and detailed descriptions about these graphs can also be found there. Sizes of these graphs are shown in Table 3.

Synthetic Graphs: We evaluate the algorithms on three kinds of synthetic graphs, all of which are generated by the graph generator GTGraph[3]. The three kinds of synthetic graphs are as follows.

- Random graphs: Random graphs with $a \times 1000$ vertices and $b \times 1000$ edges are denoted by *Random-a-b*, where a and b are integers and the $b \times 1000$ edges are added by randomly choosing a pair of vertices for each edge.

[1]Note that, TestConnect was not studied in previous works for computing k-edge connected components.

[2]`http://snap.stanford.edu/data/`

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

ID	Dataset	#Vertices	#Edges
D1	ca-GrQc	5,242	14,484
D2	ca-CondMat	23,133	93,439
D3	email-EuAll	265,214	364,481
D4	soc-Epinions1	75,879	405,740
D5	wiki-Talk	2,394,385	4,659,565
D6	amazon0601	403,394	2,443,408
D7	web-Google	875,713	4,322,051
D8	as-Skitter	1,696,415	11,095,298

Table 3: Sizes of real graphs

- PowerLaw graphs: Similar to Random-a-b, *PowerLaw-a-b* denote a PowerLaw graph with $a \times 1000$ vertices and $b \times 1000$ edges. The degree distributions of PowerLaw graphs conform with the power-law distribution.
- SSCA graphs: *SSCA-a* denote a SSCA graph with 2^a vertices. A SSCA graph contains a collection of randomly sized cliques, and then inter-clique edges are added randomly.

We generated two random graphs, Random-20-120, and Random-20-140, two power-law graphs, PowerLaw-20-120, and PowerLaw-20-140, and five SSCA graphs, SSCA-12, SSCA-14, SSCA-16, SSCA-18, and SSCA-20. The sizes of the five SSCA graphs are shown in Table 4.

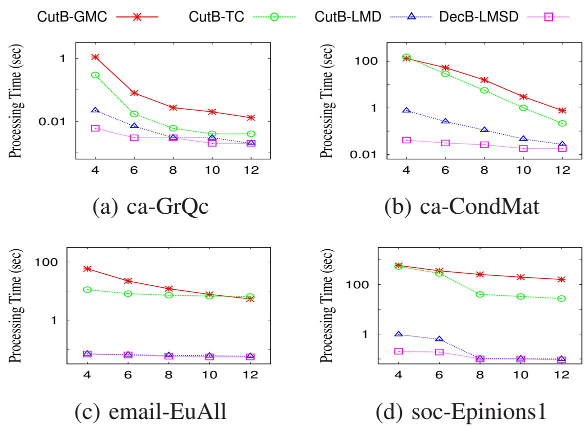
Dataset	#Vertices	#Edges
SSCA-12	4,096	24,584
SSCA-14	16,384	143,744
SSCA-16	65,536	896,759
SSCA-18	262,144	5,640,272
SSCA-20	1,048,576	35,318,325

Table 4: Sizes of SSCA graphs

For all these testings, we vary k for the connectivity requirement of k-edge connected components. Each experiment is run three times, and the average CPU time is reported here.

5.1 Decomposition-based Against Cut-based

In this test, we evaluate the effectiveness of our proposed graph decomposition-based framework against the cut-based framework studied in [20]. CutB-GMC, CutB-TC, CutB-LMD are cut-based algorithms while DecB-LMSD is a graph decomposition-based algorithm.

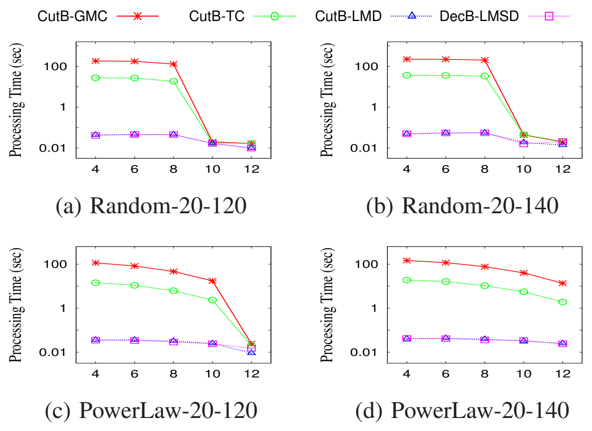


(a) ca-GrQc (b) ca-CondMat

(c) email-EuAll (d) soc-Epinions1

Figure 6: Against cut-based algorithms (Varying k)

Results on Real Graphs: We evaluate CutB-GMC, CutB-TC, CutB-LMD, and DecB-LMSD on four of the real graphs, since the other four real graphs are too large for the slow algorithms to finish in reasonable time. The processing time are shown in Fig. 6. A similar trend found in all figures of Fig. 6 is that the processing times of all four algorithms tend towards smaller when k increases. When k becomes larger, the resulting graph after removing all vertices with degrees less than k becomes smaller, therefore all algorithms run faster.[4] CutB-TC runs slightly faster than CutB-GMC on all four graphs which conforms with their time complexity differences as discussed in Section 2.2, i.e., $O(|V||E|)$ for testing connectivity versus $O(|V||E| + |V|^2 \log |V|)$ for finding global min-cut. DecB-LMSD outperforms both CutB-GMC and CutB-TC by several orders of magnitude. Although CutB-LMD runs much faster than CutB-GMC due to our optimization techniques, it is still much slower than DecB-LMSD. In Fig. 6(c), the processing times of CutB-LMD and DecB-LMSD are almost the same. The reason is that, after removing all vertices with degree less than k, the remaining graph consists of a single k-edge connected component, i.e., $h = 1$, then the performance between the cut-based algorithm and the graph decomposition-based algorithm becomes marginal on this graph.

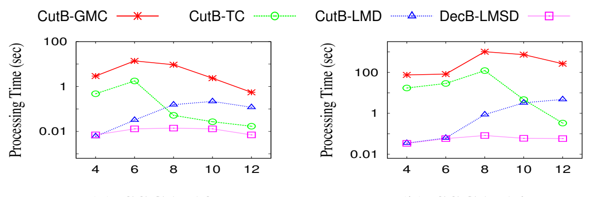


(a) Random-20-120 (b) Random-20-140

(c) PowerLaw-20-120 (d) PowerLaw-20-140

Figure 7: Against cut-based algorithms (Varing k)

Results on Random and PowerLaw Graphs: The processing times of CutB-GMC, CutB-TC, CutB-LMD, DecB-LMSD on random graphs and power-law graphs are plotted in Fig. 7. Similar to that shown in Fig. 6, DecB-LMSD outperforms both CutB-GMC and CutB-TC by more than three orders of magnitude. As shown in all figures of Fig. 7 except Fig. 7(d), when k is larger than a certain value (e.g., $k \geq 10$ in Fig. 7(a)), the processing times of all four algorithms are almost the same, due to that there is no subgraph that is k-connected, i.e., the graph is empty after removing all vertices with degree less than k. The ratio between the processing time of CutB-TC and that of CutB-GMC is almost the same for all graphs with different k values. The process times of DecB-LMSD and CutB-LMD are the same on all graphs, since the random graphs and power-law graphs tend to either have no subgraph with connectivity at least k or have only one giant subgraph with connectivity at least k. By comparing Fig. 7(a) and Fig. 7(c), Fig. 7(b) and Fig. 7(d). We can infer that power-law graphs tend to have larger connectivity compared with random graphs.



(a) SSCA-12 (b) SSCA-14

Figure 8: Against cut-based algorithms (Varying k)

[4]Note that, the technique of removing all vertices with degrees less than k is applied on all of the tested algorithms, while our optimization of efficient vertex reduction by degree is an efficient approach to realize this reduction.

Results on SSCA Graphs: Fig. 8 shows the evaluation results of the four algorithms on two SSCA graphs. As a SSCA graph contains a set of randomly sized cliques, the processing times shown in Fig. 8 have different patterns as compared with that shown in Fig. 7. In Fig. 8(a), the processing time of CutB-GMC increases when k increases from 4 to 6 as opposite to that in Fig. 6 and Fig. 7. This is because that, when k increases from 4 to 6, the size of each k-edge connected component decreases, however, the number of k-edge connected components increases as a result of the large number of cliques contained. DecB-LMSD still outperforms CutB-GMC by several orders of magnitude. However, CutB-TC also performs well for larger k, because the size of k-edge connected components becomes very small for larger k and TestConnect tends to find more balanced cut than Mas. Intuitively, TestConnect divides a non k-connected subgraph by the minimum cut between a pair of random vertices whose minimum cut values are less than k, while finding global minimum cut using Mas removes all the adjacent edges of a subgraph after contracting all vertices in it into a single super-vertex. Therefore, CutB-TC can reduce the size of each connected subgraph very quickly, while CutB-GMC divides a graph into one large and one small subgraph each time. As shown in Fig. 8, unlike those shown in Fig. 6 and Fig. 7, CutB-LMD performs much worse than DecB-LMSD, due to the larger number of k-edge connected components contained in SSCA graphs than the small real graphs, random graphs, and power-law graphs. Therefore, CutB-LMD needs to compute a global minimum cut for a lot of large subgraphs of the input graph.

Dataset	Number of components	Max size of components	h	l
ca-GrQc	10	80	2	5
ca-CondMat	14	2,020	2	7
email-EuAll	1	2,762	1	2
soc-Epinions1	1	9,337	1	3
wiki-Talk	1	47,081	1	3
amazon0601	979	8,855	3	21
web-Google	1,031	171,398	3	11
as-Skitter	9	325,486	2	9
SSCA-12	166	94	3	7
SSCA-14	359	346	5	15
SSCA-16	79	59,387	3	11
SSCA-18	30	254,996	3	9
SSCA-20	22	1,037,606	2	8

Table 5: Value of h and l of DecB-LMSD (k = 10)

The value of h and l: We test the value of h and l which contributes to the time complexity $O(h\times l\times|E|)$ of our DecB-LMSD algorithm. The values of h and l by applying DecB-LMSD on the eight real graphs and the five SSCA graphs with $k = 10$ are shown in Table 5, in which we also list out the number of k-edge connected components and the maximum size of k-edge connected components (in terms of number of vertices in it) of those graphs with $k = 10$. Among the real graphs, amazon0601 and web-Google have the largest number of k-edge connected components, and they also have large size of k-edge connected components. SSCA graphs either have a lot of small k-edge connected components (for large k) or have just a few large k-edge connected components (for small k), but not both. Despite the large number of k-edge connected components and/or the large size of k-edge connected components, both h and l in our DecB-LMSD computation are of small values. Combining this fact with our $O(h\times l\times|E|)$ time complexity, DecB-LMSD can finish in a few seconds even on large graphs.

Scalability Testing: We test the scalability of CutB-GMC, CutB-TC, CutB-LMD, and DecB-LMSD. We generate four Random graphs, Random-a-10a, and four PowerLaw graphs, PowerLaw-a-10a, with

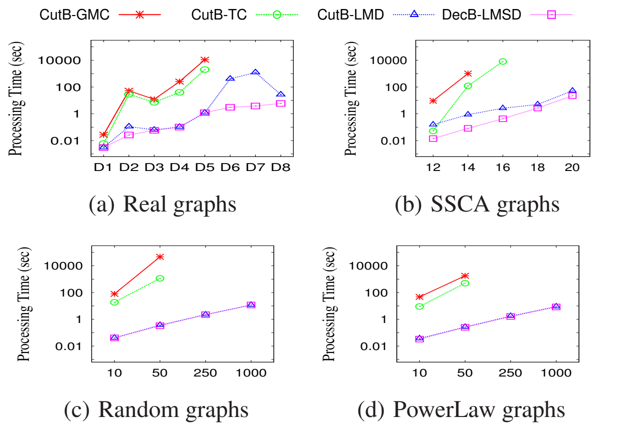


Figure 9: Scalability test (Varying graphs, k = 8)

a = 10, 50, 250, 1000, respectively. The testing results with k = 8 are shown in Fig. 9. The x-axis denotes the different graphs. The corresponding graphs in Fig. 9(b) can be inferred from the label of the x-axis with "SSCA-" omitted, e.g., 12 means the SSCA-12 graph. The exact name of the corresponding graphs in Fig. 9(a) can be found from Table 3. The x-axis of Fig. 9(c) and Fig. 9(d) denote the number of vertices ($\times10^3$). Some of the results are not plotted in Fig. 9, because the corresponding algorithms fail to terminate in five hours on the corresponding configurations. CutB-GMC and CutB-TC are not scalable to large graphs. The performance of CutB-LMD depends on the number of k-edge connected components. For example, among the eight real graphs, D5 (amazon0601) and D6 (web-Google) have large number of k-edge connected components, then CutB-LMD performs much worse than DecB-LMSD. DecB-LMSD scales almost linearly with respect to the size of input graphs as shown in Fig. 9(b), where the sizes of the five SSCA graphs increase exponentially. From Fig. 9(c) and Fig. 9(d), we see that the processing times of CutB-LMD and DecB-LMSD are almost the same, because there is only one large k-edge connected component in each of these graphs.

5.2 Evaluating Optimization Techniques

In this subsection, we evaluate the effectiveness of the proposed optimization techniques in Section 4. Note that, we also consider the data structure proposed in Section 4.2 as an optimization. Therefore, there are four optimization techniques: **L**inear data structure, early **M**erge, list **S**haring, and efficient vertex reduction by **D**egree. Let the four letters (**L**, **M**, **S**, **D**) denote the corresponding optimization applied in an algorithm, respectively. We evaluated a series of algorithms, DecB-M, DecB-LM, DecB-MSD, DecB-LMS, DecB-LMD, DecB-L, DecB-LSD. For example, DecB-LMS is an algorithm by applying the first three optimization techniques.

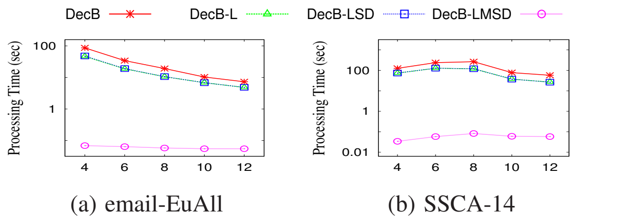
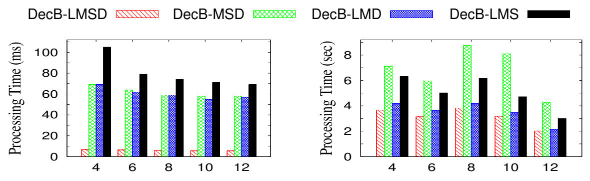


Figure 10: Testing early merge optimization(Varying k)

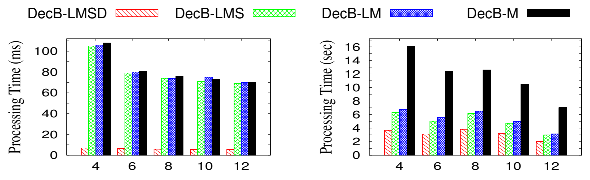
Testing the early merge optimization: We evaluate the effectiveness of the early merge optimization by comparing DecB-LMSD with other algorithms without this optimization, such as DecB, DecB-L, DecB-LSD. The results of running these algorithms on email-EuAll graph and SSCA-14 graph are reported in Fig. 10. DecB-L and DecB-LSD run slightly faster than the baseline algorithm DecB, however, both of them run much slower than DecB-LMSD by several orders of magnitude. The effectiveness of early merge optimization is due to the fact that a lot of vertex-pairs are merged in one iteration. Therefore, the number of vertices reduces rapidly after each iteration of Mas-LMS in Alg. 3 as discussed in Section 4.4, which results in a very small l. Without this optimization, for a k-connected graph $G(V, E)$, the running time would be $\Omega(|V||E|)$. In the following testings, we always include the early merge optimization.



(a) email-EuAll (b) web-Google

Figure 11: Testing L, S, D optimization techniques (Varying k)

Testing L, S, D optimization techniques: We test the effectiveness of the three optimizations, linear data structure, list sharing, and efficient vertex reduction by degree, by not considering the corresponding optimization in an algorithm while applying all other optimization techniques. The testing results are shown in Fig. 11. On the email-EuAll graph, leaving out any optimization will increase the processing time as shown in Fig. 11(a), while on the web-Google graph leaving out the list sharing optimization does not affect the performance of DecB-LMSD as shown in Fig. 11(b).



(a) email-EuAll (b) web-Google

Figure 12: Adding optimization incrementally(Varying k)

Adding optimization techniques incrementally: We test the effect of adding one optimization at a time incrementally, starting from the DecB-M algorithm. The optimization techniques are added in the following order, linear data structure, list sharing, and vertex reduction by degree, where the final algorithm is DecB-LMSD. In Fig. 12(a) we see that, the performance does not improve until all optimization techniques are added. However, in Fig. 12(b), we can see performance improvement after adding each of the three optimization techniques.

From the above evaluations, we conclude that the early merge optimization improves the performance substantially, while the other optimization techniques also improve the performance of algorithms after applying the early merge optimization.

6. CONCLUSION

In this paper, we have proposed a novel graph decomposition paradigm to iteratively decompose a graph G for computing its k-edge connected components with a very small decomposition depth h. To compute k-edge connected components efficiently based on this decomposition paradigm, we devised an efficient threshold-based graph decomposition algorithm with time complexity $O(l \times |E|)$ with a small integer l (usually $l \ll |V|$). As a result, we have improved the time complexity of an existing state-of-the-art solution of computing k-edge connected components of a graph from $O(|V|^2|E| + |V|^3 \log |V|)$ to $O(h \times l \times |E|)$, where l and h are usually bounded by a small constant in all the real and synthetic graphs we tested. As their relationships and the average case behavior of $h \times l$ are very challenging to deal with, we will put this issue in our future work. We finally conducted experiments to evaluate the performance of the proposed algorithm, and the experimental results demonstrate that our techniques outperform the existing one by several orders of magnitude.

Acknowledgements: The work was started when the first author was at CUHK, and partially done when the first and the fourth authors were at East China Normal University. The second author was partially supported by grants of the Research Grants Council of the Hong Kong SAR, China, No.418512. The fourth author was partially supported by NSFC61232006, NSFC61021004, ARC DP110102937, ARC DP120104168. The second, the fifth and the sixth authors were partially supported by ARC DP120102627.

7. REFERENCES

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

[2] V. Batagelj and M. Zaversnik. An o(m) algorithm for cores decomposition of networks. *CoRR*, cs.DS/0310049, 2003.

[3] A. A. Benczúr and D. R. Karger. Randomized approximation schemes for cuts and flows in capacitated graphs. *CoRR*, cs.DS/0207078, 2002.

[4] L. Chang, J. X. Yu, and L. Qin. Fast maximal cliques enumeration in sparse graphs. *Algorithmica*, 66(1), 2013.

[5] J. Cheng, Y. Ke, S. Chu, and M. T. Özsu. Efficient core decomposition in massive networks. In *Proc. of ICDE'11*, 2011.

[6] J. Cheng, Y. Ke, A. W.-C. Fu, J. X. Yu, and L. Zhu. Finding maximal cliques in massive networks by h*-graph. In *Proc. of SIGMOD'10*, 2010.

[7] W. S. Fung, R. Hariharan, N. J. A. Harvey, and D. Panigrahi. A general framework for graph sparsification. In *Proc. of STOC'11*, 2011.

[8] Z. Galil and G. F. Italiano. Reducing edge connectivity to vertex connectivity. *SIGACT News*, 22(1), Mar. 1991.

[9] S. Günnemann, B. Boden, and T. Seidl. Finding density-based subspace clusters in graphs with feature vectors. *Data Min. Knowl. Discov.*, 25(2), 2012.

[10] N. Hiroshi and W. Toshimasa. Computing k-edge-connected components of a multigraph (special section on discrete mathematics and its applications). *IEICE transactions on fundamentals of electronics, communications and computer sciences*, 76(4), 1993-04-25.

[11] D. W. Matula. Determining edge connectivity in 0(nm). In *Proc. of FOCS'87*, 1987.

[12] H. Nagamochi and T. Ibaraki. A linear time algorithm for computing 3-edge-connected components in a multigraph. *Japan Journal of Industrial and Applied Mathematics*, 9, 1992.

[13] A. N. Papadopoulos, A. Lyritsis, and Y. Manolopoulos. Skygraph: an algorithm for important subgraph discovery in relational graphs. *Data Min. Knowl. Discov.*, 17(1), Aug. 2008.

[14] M. Stoer and F. Wagner. A simple min-cut algorithm. *J. ACM*, 44(4), 1997.

[15] Y. H. Tsin. Yet another optimal algorithm for 3-edge-connectivity. *J. of Discrete Algorithms*, 7(1), 2009.

[16] N. Wang, J. Zhang, K.-L. Tan, and A. K. H. Tung. On triangulation-based dense neighborhood graph discovery. *Proc. VLDB Endow.*, 4(2), Nov. 2010.

[17] X. Yan, X. J. Zhou, and J. Han. Mining closed relational graphs with connectivity constraints. In *Proc. of KDD'05*, 2005.

[18] Z. Zeng, J. Wang, L. Zhou, and G. Karypis. Out-of-core coherent closed quasi-clique mining from large dense graph databases. *ACM Trans. Database Syst.*, 32(2), June 2007.

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

[20] R. Zhou, C. Liu, J. X. Yu, W. Liang, B. Chen, and J. Li. Finding maximal k-edge-connected subgraphs from a large graph. In *Proc. of EDBT'12*, 2012.