Index-based Optimal Algorithms for Computing Steiner Components with Maximum Connectivity

Lijun Chang[†], Xuemin Lin[£†], Lu Qin[‡], Jeffrey Xu Yu[§], Wenjie Zhang[†]
[†]University of New South Wales, Australia [£]East China Normal University, China
[‡]University of Technology, Sydney, Australia [§]The Chinese University of Hong Kong, China
{ljchang,lxue,zhangw}@cse.unsw.edu.au, lu.qin@uts.edu.au, yu@se.cuhk.edu.hk

ABSTRACT

With the proliferation of graph applications, the problem of efficiently computing all k-edge connected components of a graph G for a user-given k has been recently investigated. In this paper, we study the problem of efficiently computing the *steiner* component with the maximum connectivity; that is, given a set q of query vertices in a graph G, we aim to find the maximum induced subgraph g of G such that g contains q and g has the maximum connectivity, where g is denoted as SMCC. To accommodate online query processing, we present an efficient algorithm based on a novel index such that the algorithm runs in linear time regarding the result size; thus, the algorithm is optimal since it needs at least linear time to output the result. Moreover, in this paper we also investigate variations of the above problem. We show that such a problem with the constraint that the size of the SMCC is not smaller than a given size can also be solved in linear time regarding the result size (thus, optimal). We also show that the problem of computing the connectivity (rather than the graph details) of SMCC can be solved in linear time regarding the query size (thus, optimal). To build the index, we extend the techniques in [7] to accommodate batch processing and computation sharing. To efficiently support the applications with graph updates, we also present novel increment techniques. Finally, we conduct extensive performance studies on large real and synthetic graphs, which demonstrate that our index-based algorithms significantly outperform baseline algorithms by several orders of magnitude and our indexing algorithms are efficient.

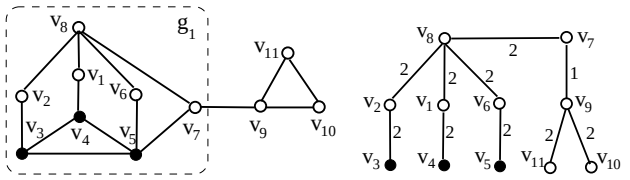
Categories and Subject Descriptors
H.2.8 [**Database Applications**]: Data mining; G.2.2 [**Graph Theory**]: Graph algorithms

Keywords
Steiner Maximum-Connected Component; k-Edge Connected Component; Maximum Spanning Tree; Dynamic Graph

1. INTRODUCTION

In many real applications, data and their relationships can be modeled as a graph $G = (V, E)$, where vertices in V represent entities of interest and edges in E represent relationships between entities. With the proliferation of graph applications, such as social networks, information networks, web search, collaboration networks, E-commerce networks, communication networks, and biology, research efforts have been devoted towards many fundamental problems in managing and analyzing graph data. Among them, the problem of computing all *k-edge connected components* of a graph has been recently studied in [4, 7, 31, 34] for a given k. A k-edge connected component of a graph G is a maximal vertex-induced subgraph g of G such that g is *k-edge connected* (i.e., the resulting graph is still connected after the removal of any $(k-1)$ edges from g). For example, the graph G in Figure 1(a) is 1-edge connected, and g_1 (the left subgraph of G) is a 2-edge connected component.

Computing k-edge connected components has many applications. For example, in social networks (e.g., Facebook), a *cohesive* block (a community) of a graph G is defined as a connected component g of G where the connectivity of g is referred to as the *cohesiveness* of the cohesive block g [30]. Computing connected components with high connectivity may also be used to identify the closely related entities to provide useful information for social behavior mining [2]. In a collaboration network (e.g., DBLP), a highly connected component may be a group of researchers with similar research interests. In an E-commerce network (e.g., eBay), a highly connected subgraph may be a group of products that are interested by customers with similar flavors. 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 and robust detection of communication networks [4, 7, 29, 33].



(a) Graph (b) Maximum spanning tree

Figure 1: Example

Computing Steiner Maximum-Connected Components. In the above applications, users may often want to find a subgraph with the maximum connectivity that contains a given set q of query vertices. There could be many such subgraphs with the maximum connectivity. In this paper, we focus on computing the maximum induced subgraph with the maximum connectivity among all subgraphs that contain q, namely, *Steiner Maximum-Connected Component* (SMCC). We also investigate the problem of computing the SMCC with its number of vertices not smaller than a given L, namely SMCC_L. Finally, instead of computing SMCC, we also

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.

ACM 978-1-4503-2758-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2723372.2746486.

investigate the problem of computing the connectivity of SMCC, namely the *steiner-connectivity* of q.

Example 1.1: Assume that in the graph G in Figure 1(a), the set q of query vertices is $\{v_3, v_4, v_5\}$. Regrading the given q, below are the results for the above three problems. The SMCC is g_1. The SMCC_L for $L = 9$ is the entire graph G; that is, the SMCC containing q with at least 9 vertices is G. The steiner-connectivity of q is 2, which is the connectivity of the SMCC g_1. □

Applications. The above problems have many applications.
1) Potential Customer Prediction. Predicting potential customers for a new product is essential for marketing. Social networks (e.g., Facebook) provide a good way to do the suggestion by utilizing the friend relationship among users. Given a set q of users who are already interested in a certain product, it is highly possible that other users who are highly connected to q and also highly connected to each other will be interested in the product. This is because highly connected users are likely to belong to the same social cluster [19] and thus share similar interests. Therefore, such customers can be found by computing the SMCC or SMCC_L of q (i.e., the most highly connected component containing q).
2) Product Promotion. In an E-commerce network (e.g., eBay), a sales manager may want to promote some products that have high potential to be popular. Those products that are in the same highly associated group as the set of hot products are good candidates for the promotion, while a highly associated group is usually identified as a graph component with high connectivity [31]. Therefore, such products can be obtained by computing SMCC or SMCC_L by considering the set of hot products as a query q.
3) Research Team Assembling. Several key researchers may want to assemble a research team to work on a big research project. Intuitively, researchers in a good team should have close collaboration such that the overall communication cost is small [22]. Therefore, the researchers who are highly connected to the initiators and also highly connected to each other in a collaboration network (e.g., DBLP) are good candidates to be invited to the team [16]. Such a problem can be solved by computing SMCC or SMCC_L with the set of key researchers as the query q. Moreover, the steiner-connectivity of the key researchers indicates how strong they are connected to each other (possibly via other researchers).

Challenges. To the best of our knowledge, this is the first work to investigate the problem of efficiently computing SMCC (and also its variant, SMCC_L). Clearly, the entire graph or the subgraph induced by q is not always the solution. An immediate solution is to enumerate all induced subgraphs, containing q, of G to test its connectivity, and then choose the maximum induced subgraph with the maximum connectivity as the result. Nevertheless, such an algorithm is too expensive since it may involve enumerating an exponential number of induced subgraphs.

Alternatively, we could adopt the state-of-the-art techniques in [4] or [7] to compute all k-edge connected components of G for a given k while decreasing k from $|V|$ to 1. Then, we choose the first connected component that contains q, as the result; this component would have the maximum connectivity. However, this approach needs to traverse the entire graph G several times, which takes excessive long time when G is large.

Motivated by the above, in this paper we aim to develop efficient techniques to compute SMCC based on an iterative expansion paradigm to only include vertices that are in the result graph one by one till finding the SMCC. By doing so, we can avoid scanning the whole graph but only involve the necessary vertices. However, this is not trivial. Regarding the graph in Figure 1(a) with $q = \{v_3, v_4, v_5\}$, the result graph has to contain g_3, the triangle with the vertex set q. While the expansion from g_3 to g_1 needs to have g_2 as an intermediate result where g_2 is the subgraph induced by the vertices $\{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$, it seems hard to justify the expansion to g_2 from g_3 and then to g_1 since g_3 is 2-edge connected but g_2 is only 1-edge connected; this is the challenge. Moreover, in many cases vertices in q are not directly connected to each other; for example, $q = \{v_3, v_1, v_6\}$ in Figure 1(a) may also be a query. This seems making the challenge harder, since we also need to do expansions for finding paths to connect the vertices in q together.

Our Approaches and Contributions. Assume that the steiner-connectivity of each pair of vertices in G is pre-computed and denoted by $sc(\{u, v\})$. In this paper, we show that we can compute the SMCC of q by 1) computing the steiner-connectivity $sc(q)$ of q, and 2) including all vertices in G whose steiner-connectivity with q is no less than $sc(q)$ into the result. To efficiently implement the above idea, we prove that $sc(q) = \min_{v \in q} sc(\{v_0, v\})$ with v_0 being an arbitrary vertex in q. This is the basic idea of our techniques. Such an algorithm can be executed in linear time regarding the number of vertices in the result if for each vertex v in G, other vertices $\{v' \in V\}$ are sorted over the values of $sc(\{v, v'\})$. However, the space complexity is $O(|V|^2)$ which is prohibitively large.

To resolve the space issue, we propose a novel compact tree-structure index to preserve the steiner-connectivity information. In particular, we show that the maximum spanning tree T, over the transitive closure of G with the steiner-connectivities as the edge weights, gives enough information for our computation. As an example, Figure 1(b) demonstrates such a maximum spanning tree T for the graph in Figure 1(a) where the number on each edge is the steiner-connectivity of the two end-vertices.

Then, we show that the steiner-connectivity of q equals the minimum weight in the subtree T_q of T, where T_q is the minimal connected subtree of T that contains q. For example, for $q = \{v_3, v_4, v_5\}$ in Figure 1(b), T_q is the subtree spanning $\{v_1, v_2, v_3, v_4, v_5, v_6, v_8\}$ and then $sc(q) = 2$. Thus, the SMCC of q can be obtained by adding vertices one by one, starting from any vertex in q, in a BFS fashion on T if its corresponding edge's weight in T is at least $sc(q)$. For example, regarding the spanning tree T in Figure 1(b) and $q = \{v_3, v_4, v_5\}$, the algorithm starts from v_4 to add the vertices $v_1, v_8, v_2, v_6, v_7, v_3, v_5$ one by one in a BFS fashion, where the subtree spanning v_9, v_{10}, and v_{11} is pruned since $sc(\{v_1, v_9\}) = 1$. It is immediate that such an algorithm runs in liner time regarding the number of vertices outputted if we order the neighbours of each vertex in the tree based on their values of steiner-connectivities.

To improve the efficiency of computing the steiner-connectivity of q, we further develop a new tree index, by which we achieve the time complexity of $O(|q|)$ for computing the steiner-connectivity. Moreover, we also show that SMCC_L can be computed by a prioritized search on T with time complexity linear to the result size.

To build such an index, the maximum spanning tree T, we show that we only need to pre-compute the steiner-connectivities for all edges in G. By extending the techniques in [7] for batch processing and computation sharing, this can be conducted in time $O(\alpha(G) \cdot h \cdot l \cdot |E|)$, where $\alpha(G)$ is the "arboricity" of G and is bounded by (usually much smaller than) $\sqrt{|E|}$ [10], and h and l are usually bounded by small constants [7]. Moreover, we also propose incremental techniques to efficiently update the steiner-connectivities and our index structure when the graph changes. In particular, we show that to update the steiner-connectivities when an edge (u, v) is inserted or deleted, we can restrict our computation to the SMCC of $\{u, v\}$; moreover, we can contract every k-edge connected component with $k = sc(q) + 1$ in the SMCC into a super-vertex. This greatly reduces the index maintenance cost.

Our primary contributions consist of the following two parts.

- *Optimal Query Processing Algorithms.*
  We propose index-based optimal algorithms for computing

SMCC, SMCC$_L$, and steiner-connectivity; that is, the algorithms are in linear time regarding the query and result sizes.

- *Efficient Index Construction and Maintenance Techniques.* 1) For index construction, we propose an efficient algorithm to compute the steiner-connectivity for all edges in G in time $O(\alpha(G) \cdot h \cdot l \cdot |E|)$. We show that the maximum spanning tree is enough to preserve the steiner-connectivity for all edges.

  2) We also propose efficient incremental techniques to update the steiner-connectivity and our index structure when the graph changes. To do this, we identify important properties to restrict our computation locally to a small subgraph.

We conduct extensive empirical studies on large real and synthetic graphs. The empirical studies confirm that the proposed index-based algorithms significantly outperform baseline algorithms by several orders of magnitude, and demonstrate that our indexing techniques can construct and maintain the indexes very efficiently.

Organization. The rest of the paper is organized as follows. A brief overview of related work is given below. Section 2 gives the definitions of the studied queries, and Section 3 presents baseline algorithms by using the existing techniques. We propose index-based optimal query processing techniques in Section 4. Efficient techniques for index construction and maintenance are developed in Section 5. Experimental results are reported in Section 6. We discuss extensions of our techniques to process other queries and possible ways to conduct external-memory computation in Section 7, and give a conclusion in Section 8. *Proofs are omitted due to space limits and can be found in Section A.3 in the Appendix.*

Related Work. We categorize the related works as follows.

Computing k-Edge Connected Components. As discussed in the challenge part, we can compute SMCC by extending the existing techniques for computing k-edge connected components. In the literature, there are three approaches for computing k-edge connected components of a graph; that is, cut-based approach [25, 31, 34], decomposition-based approach [7], and randomized approach [4]. As the decomposition-based approach has a time complexity of $O(h \cdot l \cdot |E|)$ where h and l are usually bounded by small constants, extending it to compute SMCC takes $O(|V| \cdot h \cdot l \cdot |E|)$ time which is time-consuming; we further discuss these techniques in Section 3. In this paper, we propose optimal algorithms for computing SMCC; that is, our running time is linear to the output size.

Online Community Search. Given a set q of query vertices and a graph G, the problem of online community search that computes the communities in G containing q has been studied recently. Different semantics for community search have been studied; for example, local modularity based community search [11], k-core based community search [27, 14], k-truss based community search [20], and α-adjacency γ-quasi-k-clique based community search [13]. Nevertheless, due to inherent different problem definitions, none of these techniques can be used to compute SMCC or SMCC$_L$.

Dense Subgraph Extraction. Efficient techniques for computing all maximal *cliques* and *quasi-cliques* of a graph are presented in [6, 9] and [32], respectively. Problems of efficiently computing other dense subgraphs, including k-core [8], DN-subgraph [29], triangle k-core motifs [33], etc., have also been recently investigated. Nevertheless, due to inherently different problem natures, these techniques are inapplicable to compute SMCC or SMCC$_L$.

Edge-Connectivity. Efficiently computing edge-connectivities between vertex-pairs has been studied in graph theory [17], which can be computed by the maximum flow techniques [12]. The state-of-the-art algorithms compute exact maximum flow in $O(|V||E|)$ time [24] and approximate maximum flow in almost linear time to $|E|$ [21, 26]. To efficiently process vertex-to-vertex edge-connectivity queries, index structures have also been developed in [1] and [18]. Nevertheless, due to inherently different formulations, these techniques cannot be used to compute steiner-connectivities.

2. COMPUTING STEINER MAXIMUM-CONNECTED COMPONENTS

In this paper, we focus on an *undirected graph* $G = (V, E)$ [17], 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 in G by $|V|$ and $|E|$, respectively. Given a vertex subset $V_s \subseteq V$, the *vertex-induced subgraph* $G[V_s]$ by V_s is a subgraph $G[V_s] = (V_s, E_s)$ of G with V_s as its vertex set such that E_s consists of only the edges in G with both endpoints in V_s; that is, $G[V_s] = (V_s, \{(u, v) \in E \mid u, v \in V_s\})$.

Definition 2.1: (k-edge Connected [17]) A graph G is *k-edge connected* if the remaining graph is still connected after the removal of any $(k-1)$ edges from G. □

The *edge-connectivity* of a graph is the largest k for which the graph is k-edge connected. In the following, for presentation simplicity we refer edge-connectivity as connectivity.

Definition 2.2: (Steiner Maximum-Connected Component) Given a set q of vertices in a graph G, we define the *steiner maximum-connected component* in G of q, denoted SMCC, as the maximum induced subgraph with the maximum connectivity among all subgraphs of G that contain q. □

We call the connectivity of the SMCC of q as the *steiner-connectivity* of q, denoted $sc(q)$. SMCC is related to k-edge connected components defined below.

Definition 2.3: (k-edge Connected Component [4, 7]) Given a graph G, a subgraph g of G is a *k-edge connected component* of G if 1) g is k-edge connected, and 2) any super-graph in G of g is not k-edge connected. □

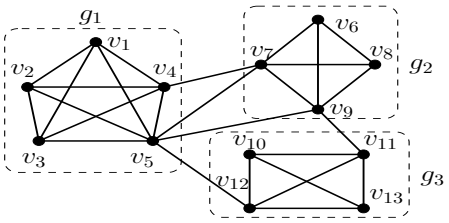


Figure 2: An example graph

A k-edge connected component is a maximum vertex-induced subgraph. It is easy to see that the SMCC of q is the k-edge connected component of G containing q with the maximum k. For example, the graph G in Figure 2 is 2-edge connected, the subgraph g_1 is a 4-edge connected component, and g_3 is a 3-edge connected component. However, g_2 is not a 3-edge connected component, since $g_1 \cup g_2$ is also 3-edge connected; $g_1 \cup g_2$ is a 3-edge connected component. Here, $g_1 \cup g_2$ denotes the union of g_1, g_2, which also includes the edges between vertices in g_1 and vertices in g_2 [17]. Therefore, the SMCC of $\{v_1, v_4\}$ is g_1 with $sc(\{v_1, v_4\}) = 4$, and the SMCC of $\{v_1, v_4, v_7\}$ is $g_1 \cup g_2$ with $sc(\{v_1, v_4, v_7\}) = 3$.

Definition 2.4: (SMCC with Size Constraint ($\geq L$)) Given a set q of vertices in a graph G and a number L, we define the *SMCC with size constraint* ($\geq L$) in G of q, denoted SMCC$_L$, as the SMCC containing q with the number of vertices not smaller than L. □

For example, in Figure 2, the SMCC$_L$ of $\{v_1, v_4\}$ with $L = 4$ is g_1, while the SMCC$_L$ of $\{v_1, v_4\}$ with $L = 6$ is $g_1 \cup g_2$.

Problem Statement. Given a set q of query vertices in a graph G and possibly a number L, we study the following three queries:

- *SMCC Query* — compute the SMCC of q;
- *SMCC$_L$ Query* — compute the SMCC$_L$ of q;
- *Steiner-Connectivity Query* — compute $sc(q)$.

Note that, in this paper we focus on computing the set of vertices for SMCC and SMCC$_L$ queries, while the subgraph details can be obtained as induced subgraphs. In the following, without loss of generality, we assume that the input graph G is connected and $|q| \geq 2$. Note that, if $|q| = 1$ (assume $q = \{v\}$), then we can replace q with a new query $q' = \{v, v'\}$, where $v' = \arg\max_{(v,v'')\in E} sc(v, v'')$; it is easy to verify that q and q' have the same query result.

3. BASELINE SOLUTIONS

As shown in Section 2, the SMCC of q is the k-edge connected component of G containing q with the maximum k. Thus, we can process an SMCC query by computing k-edge connected components of G for all different values of k, as follows. We decrease the value of k from $|V|$ to 1, compute the corresponding k-edge connected components of G, and terminate the computation once there is a k-edge connected component containing q which then is the SMCC of q. The pseudocode is shown in Algorithm 1 below.

Algorithm 1: BaseLine

Input: A graph $G = (V, E)$, and a set q of vertices
Output: The SMCC of q

```
1 for k ← |V| to 1 do
2     φ_k(G) ← ComputeKECCs(G, k);          /* Compute k-edge
      connected components of G */;
3     if there is a subgraph g in φ_k(G) containing q then
4         return g as the SMCC of q;
```

It is immediate that Algorithm 1 correctly computes the SMCC of q, given that ComputeKECCs correctly computes all k-edge connected components of G. In the literature, there are two state-of-the-art algorithms for computing k-edge connected components of a graph: one is an exact algorithm in [7], denoted KECCs-Exact, which is also briefly presented in Section A.5, and the other is a Monte Carlo randomized algorithm in [4], denoted KECCs-Random. Both algorithms have shown to be superior than the cut-based algorithms in [25, 31, 34]. In this paper, we adopt both algorithms in Algorithm 1, and denote the corresponding algorithms for processing SMCC queries as SMCC-BLE and SMCC-BLR, respectively.

Time Complexities. Here, we analyse the time complexities.

Time Complexity of SMCC-BLE. As shown in [7] (and Section A.5), the time complexity of KECCs-Exact is $O(h \cdot l \cdot |E|)$, where h and l are usually bounded by small constants for real graphs. Therefore, the time complexity of SMCC-BLE is $O(|V| \cdot h \cdot l \cdot |E|)$.

Time Complexity of SMCC-BLR. As shown in [4], the time complexity of KECCs-Random is $O(t \cdot |E|)$, where t is the number of iterations; it is proved in [4] that KECCs-Random, with high probability, can correctly compute all k-edge connected components of a graph in $O(\log^2 |V|)$ iterations (i.e., $t = O(\log^2 |V|)$). Therefore, the time complexity of SMCC-BLR is $O(|V| \cdot t \cdot |E|)$.

Other Queries. Note that, Algorithm 1 can be immediately used to compute the steiner-connectivity of q; that is, at Line 4, we return k instead of g. Similarly, it is easy to prove that the SMCC$_L$ of q is the k-edge connected component of G with the maximum k that contains q and has no less than L vertices. Thus, Algorithm 1 can be used to process SMCC$_L$ queries as well, by returning the first g that satisfies Line 3 and also has no less than L vertices.

4. OPTIMAL QUERY PROCESSING

In this section, we propose index-based optimal algorithms for computing the SMCC, the SMCC$_L$, and the steiner-connectivity for a set of query vertices, while indexing techniques will be presented in Section 5. Firstly, we present the general idea and an overview of our algorithms in Section 4.1 and develop our index structure in Section 4.2. Then, we propose index-based optimal algorithms for processing steiner-connectivity queries, SMCC queries, and SMCC$_L$ queries in Sections 4.3, 4.4, and 4.5, respectively.

4.1 General Idea and An Overview

We present the following lemma to characterize the SMCC of q, which also illustrates the general idea of our algorithm.

Lemma 4.1: *Given a set q of query vertices, the SMCC of q is the maximal set of vertices such that each vertex v in it satisfies $sc(\{v_0, v\}) \geq sc(q)$, where v_0 is an arbitrary vertex chosen from q and $sc(q)$ is the steiner-connectivity of q.* □

Here, $sc(\{v_0, v\})$ is the steiner-connectivity of $\{v_0, v\}$, and we abbreviate it as $sc(v_0, v)$ in the following for ease of presentation. From Lemma 4.1, we can see that the SMCC of q can be obtained as $\{v \in V \mid sc(v_0, v) \geq sc(q)\}$ once we have computed the steiner-connectivity of q. Moreover, the steiner-connectivity of q can be computed based on the lemma below.

Lemma 4.2: *Given $q = \{v_0, \ldots, v_{|q|-1}\}$, the steiner-connectivity of q, $sc(q)$, is equal to the minimum steiner-connectivity between v_0 and any vertex in q, where v_0 is an arbitrarily chosen vertex from q; that is, $sc(q) = \min_{1\leq i\leq |q|-1} sc(v_0, v_i), \forall v_0 \in q$.* □

Note that, $sc(\{v_0\}) \geq sc(q)$. Therefore, we can simplify the equation in Lemma 4.2 as $sc(q) = \min_{v\in q} sc(v_0, v), \forall v_0 \in q$.

Overview. Following from Lemma 4.1 and Lemma 4.2, we can process any SMCC query in two steps by Algorithm 2. We first compute the steiner-connectivity of q based on Lemma 4.2 (Line 1), and then compute the SMCC of q based on Lemma 4.1 and the computed $sc(q)$ (Line 2).

Algorithm 2: SMCC-Overview

Input: A graph $G = (V, E)$, and a set $q = \{v_0, \ldots, v_{|q|-1}\}$ of vertices
Output: The SMCC of q

```
1 sc(q) ← min_{v∈q} sc(v_0, v);
2 V_q ← {v ∈ V | sc(v_0, v) ≥ sc(q)};
3 return V_q;
```

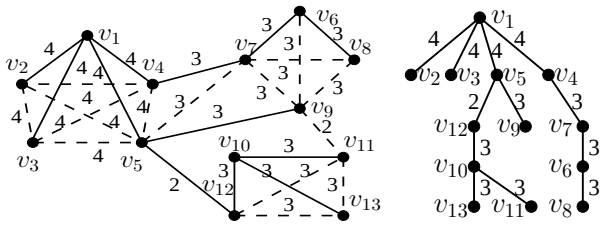
A Possible but Impractical Implementation. One possible implementation is pre-computing and storing the steiner-connectivity $sc(u, v)$ for every pair of vertices in G; that is, we store the transitive closure of G with each edge (u, v) in it carrying a weight $sc(u, v)$. Then, Line 1 runs in $O(|q|)$ time. Moreover, if we pre-sort all edges (u, v) regarding the same vertex u by their weights, then Line 2 runs in $O(|V_q|)$ time. Consequently, such an implementation of Algorithm 2 runs in $O(|V_q|)$ time, where V_q is the result.

However, the space complexity will be $O(|V|^2)$, which is prohibitively large. For example, even for a moderate-sized graph with 3×10^6 vertices, the storage space will be over 10TB. Therefore, this implementation is not practical for large graphs.

4.2 Observations and Index Structure

In this subsection, *we propose a compact index structure to preserve all the steiner-connectivity information while still enabling fast query processing.* For developing our index structure, we first define the connectivity graph and present an important property of steiner-connectivity based on the connectivity graph.

Definition 4.1: (Connectivity Graph) Given a graph $G = (V, E)$, the *connectivity graph* of G is a *weighted undirected graph* $G_c = (V, E, w)$ with the same set of vertices and edges as that of G. Each edge (u, v) in G_c carries a weight $w(u, v)$, which is the steiner-connectivity $sc(u, v)$ of $\{u, v\}$ in G. □



(a) Connectivity graph G_c (b) MST T

Figure 3: Connectivity graph and MST

The corresponding connectivity graph to the graph in Figure 2 is shown in Figure 3(a) (including edges indicated by both solid lines and dashed lines), where weights of edges are also shown. Connectivity graph has the following property.

Lemma 4.3: *Given vertex u and vertex v in a connectivity graph G_c, let $\mathcal{P}_{u,v}$ denote the set of all simple paths between u and v in G_c, and define the weight $w(P)$ of a path P as the minimum edge weight in P (i.e., $w(P) = \min\{w(u', v') \mid (u', v') \in P\}$), then the steiner-connectivity of $\{u, v\}$ is $sc(u, v) = \max_{P \in \mathcal{P}_{u,v}} w(P)$.* □

Lemma 4.3 illustrates the key observation for computing steiner-connectivities based on the connectivity graph. For example, for computing $sc(v_1, v_7)$ in Figure 3(a), the path (v_1, v_4, v_7) has the maximum weight among all paths between v_1 and v_7; thus $sc(v_1, v_7) = w(v_4, v_7) = 3$. Although this is not an efficient approach, we are able to develop a compact index structure based on the key observation in Lemma 4.3, in the following.

Index Structure: MST. Given the connectivity graph $G_c = (V, E, w)$ of a graph G, we construct a compact *tree-structured index* T, which is the maximum spanning tree (MST) of G_c; that is, the index is a weighted tree where each tree edge has a steiner-connectivity regarding the two end-vertices.

Definition 4.2: (Maximum Spanning Tree [17]) Given a connected, undirected graph G, a *spanning tree* of G is a subgraph of G that is a tree and contains all vertices of G. A *maximum spanning tree* of G is the spanning tree with the maximum total weight. □

The MST has the nice property that it explicitly stores the path with maximum weight for every pair of vertices as proved below.

Lemma 4.4: *Given any MST T constructed from the connectivity graph G_c, the unique path P in T between vertex u and vertex v has the maximum weight (see Lemma 4.3) among all paths between u and v in G_c. Thus, $sc(u, v) = \min_{(u',v') \in P} \lambda_T(u', v')$ where $\lambda_T(u', v')$ denotes the weight of edge (u', v') in T.* □

For example, the MST T of the graph in Figure 3(a) consists of the edges indicated by solid lines and is reillustrated in Figure 3(b). The path in T between v_1 and v_7 is (v_1, v_4, v_7) which is the path with the maximum weight among all paths in G_c between v_1 and v_7.

Index Storage. Following from Lemmas 4.2 and 4.4, we can see that any MST T preserves all the steiner-connectivity information; that is, for any $q \subset V$, we can compute $sc(q)$ just based on any MST T. Therefore, we can store any MST T and use it to process the queries we study in this paper. Note that, the connectivity graph can be simply stored by adding the weight on each edge of the original graph, where the weight of an edge is the steiner-connectivity between its two end-vertices.

The size of the MST index is $O(|V|)$. This provides a possibility to process the studied queries in main memory; that is, we store the MST in main memory. Nevertheless, in Section 7 we also discuss possible structures of disk-based index and possible ways to conduct external-memory computation.

4.3 Optimally Processing Steiner-Connectivity Queries

In this subsection, we propose an index-based optimal algorithm for processing steiner-connectivity queries. Firstly, we present the following lemma for computing the steiner-connectivity of q which directly follows from Lemma 4.2 and Lemma 4.4.

Lemma 4.5: *Given a set q of query vertices, the steiner-connectivity of q is equal to the minimum edge weight in the subtree T_q of T, where T_q is the minimal connected subtree of T that contains all vertices of q.* □

Intuitively, T_q is formed by the set of paths in T between $v_0 \in q$ and every other vertex in q. Following from Lemma 4.5, the pseudocode for processing steiner-connectivity queries is shown in Algorithm 3. Given a set q of vertices, we first obtain the subtree T_q of T, and then report the minimum edge weight among all edges in T_q as the steiner-connectivity of q.

Algorithm 3: SC-MST

Input: A MST T, and a set q of vertices
Output: The steiner-connectivity of q
1 Compute the subtree T_q;
2 **return** $sc(q) = \min_{(u,v) \in T_q} \lambda_T(u, v)$;

Implementation and Time Complexity. A naive implementation of Algorithm 3 by BFS or DFS [12] would require $O(|V|)$ time to get the subtree T_q for q, which is too slow. We demonstrate that Algorithm 3 can be implemented in $O(|T_q|)$ time in the following. Obviously, Line 2 runs in $O(|T_q|)$ time. Therefore, we only need to describe how to obtain the subtree T_q (i.e., Line 1) in $O(|T_q|)$ time.

Definition 4.3: (Lowest Common Ancestor [3]) The *Lowest Common Ancestor (LCA)* of two vertices, u and v, in a rooted tree [17], denoted $lca(u, v)$, is defined as the vertex that is farthest to the root and has both u and v as its descendants (where a vertex is allowed to be a descendant of itself). □

Similarly, we can define the LCA of a set q of vertices, denoted $lca(q)$. Given the MST T, we make it a rooted tree by choosing an arbitrary vertex to be the root. Then, it is easy to see that T_q consists of all edges in the paths from every vertex in q to $lca(q)$. Moreover, assume $q = \{v_0, v_1, \ldots, v_{|q|-1}\}$, and let $lca_i = lca(v_0, \ldots, v_i)$, then $lca_i = lca(lca_{i-1}, v_i), \forall 1 \le i \le |q| - 1$ where $lca_0 = v_0$. Thus, we can first compute lca_1, then lca_2, and so forth; finally, $lca_{|q|-1}$ is $lca(q)$.

To efficiently compute LCA, we firstly preprocess the rooted tree T: for each vertex v in T, we store its parent $p(v)$ and its level number $l(v)$, where the level number of the root vertex is 0 and $l(v) = l(p(v)) + 1$ for other vertices. Then, given u and v, we can obtain $lca(u, v)$ by traversing u and v to their ancestor vertices according to their level numbers and by following $p(\cdot)$; that is, starting from $u' (= u)$ and $v' (= v)$, each time we traverse the vertex with larger level number to its parent (i.e., $u' \leftarrow p(u')$ if $l(u') > l(v')$ and $v' \leftarrow p(v')$ otherwise), and $lca(u, v)$ is obtained as u' when u' and v' are the same. Moreover, for computing $lca_i(lca_{i-1}, v_i)$, once v_i reaches a vertex that has already been visited when computing $lca_1, \ldots, lca_{i-1}$, we can conclude that $lca_i(lca_{i-1}, v_i) = lca_{i-1}$. Finally, T_q is obtained by including all the visited edges. It is easy to verify that the running time is $O(|T_q|)$. We show the psudocode in Algorithm 10 in Section A.1 in the Appendix.

Example 4.1: Suppose $q = \{v_3, v_{13}, v_{11}\}$, where the MST T is shown in Figure 3(b) with v_1 as the root. For computing $lca(v_3, v_{13})$, we traverse from v_3 and v_{13} to their ancestor vertices until reaching v_1; thus $lca(v_3, v_{13}) = v_1$. Now, we compute $lca(lca(v_3, v_{13}), v_{11})$; firstly, we traverse v_{11} to its parent v_{10} which has been visited be-

fore, thus we conclude that $lca(q) = lca(v_3, v_{13}) = v_1$. Therefore, T_q consists of all the visited edges, and $sc(q) = \lambda_T(v_5, v_{12}) = 2$. □

The above algorithm, Algorithm 3, is efficient, however not optimal. We further propose a novel technique to preprocess the MST T to generate a new tree such that a steiner-connectivity query q can be processed in $O(|q|)$ time; thus, the algorithm is optimal. Details are presented in Section A.2 in the Appendix.

4.4 Optimally Processing SMCC Queries

Given the steiner-connectivity of q, the SMCC of q can be obtained as $\{v \in V \mid sc(v_0, v) \geq sc(q)\}$ following Lemma 4.1. Given the MST T, a naive approach would be computing $sc(v_0, v)$ for every vertex v in G based on Lemma 4.4. However, this is time-consuming. In this subsection, we propose an optimal algorithm for processing SMCC queries. Firstly, we present the lemma below that directly follows from Lemma 4.1 and Lemma 4.4.

Lemma 4.6: *Given the MST T and the steiner-connectivity $sc(q)$ of q, the SMCC of q consists of all vertices that are reachable from v_0 through edges with weights no less than $sc(q)$, where v_0 is an arbitrary vertex in q.* □

The pseudocode for optimally processing SMCC queries is shown in Algorithm 4. We first compute the steiner-connectivity $sc(q)$ of q, and then obtain the SMCC of q by conducting a BFS starting from any vertex v_0 in q and visiting only edges with weights at least $sc(q)$. Note that, the BFS at Line 2 actually computes the $sc(q)$-edge connected component of G containing v_0.

Algorithm 4: SMCC-OPT

```
Input: A MST T, and a set q of query vertices
Output: The SMCC of q
1 Compute the steiner-connectivity sc(q) of q;   /* By invoking Algorithm 3 */;
2 Conduct a BFS on T starting from any vertex v0 in q and visiting only
  edges with weights at least sc(q); let Vq be the set of visited vertices;
3 return Vq;
```

Implementation and Time Complexity. Algorithm 4 can be implemented in $O(|V_q|)$ time as follows. Firstly, Line 1 can be implemented in $O(|T_q|)$ (see Section 4.3) time by invoking Algorithm 3. Secondly, Line 2 can be implemented in $O(|V_q|)$ time, if we store T in the form of adjacency lists and organize edges in each adjacency list in non-increasing weight order. Moreover, it is obvious that T_q is a subset of V_q (i.e., $T_q \subseteq V_q$). Thus, the time complexity of Algorithm 4 is $O(|V_q|)$, which is optimal since V_q is the result.

Example 4.2: Consider a query $q = \{v_1, v_4, v_5\}$ on the graph in Figure 2 where the MST T is shown in Figure 3(a). Firstly, we obtain that $sc(q) = 4$, since $T_q = \{(v_1, v_4), (v_1, v_5)\}$ and $\lambda_T(v_1, v_4) = \lambda_T(v_1, v_5) = 4$. Secondly, we conduct a BFS on the MST T starting from v_1 and visiting only edges with weights at least 4; during the BFS, we visit vertices $\{v_1, \ldots, v_5\}$, which is the SMCC of q. □

4.5 Optimally Processing SMCC$_L$ Queries

With similar ideas to Algorithm 4, we can process an SMCC$_L$ query by trying different connectivity values k, and conducting BFS for each k. That is, we decrease k from $sc(q)$ to 1, conduct BFS on T by starting from v_0 to obtain the k-edge connected component of G containing v_0, and return the component once it has at least L vertices. However, this is not an efficient approach.

We propose an optimal algorithm for computing SMCC$_L$ based on the property that *each k-edge connected component of G is entirely contained in a $(k-1)$-edge connected component of G*. Thus, instead of computing k-edge connected components of G from scratch for each different value of k, we can gradually compute the k-edge connected components of G containing v_0 for different values of k by conducting a prioritized search on T starting from v_0. The pseudocode is shown in Algorithm 5 below.

Algorithm 5: SMCC$_L$-OPT

```
Input: A MST T, a set q of query vertices, and a number L
Output: The SMCC_L of q
1  Vq ← {v0};   /* v0 is an arbitrary vertex in q */;
2  k ← 0; /* k is a lower bound connectivity of SMCC_L */;
3  Q ← {the first adjacency edge in T of u};
4  while the maximum key in Q is not smaller than k do
5      (u, v) ← pop the edge with maximum weight from Q;
6      Push the next adjacency edge of u into Q;
7      if v ∉ Vq then
8          Vq ← Vq ∪ {v};
9          Push the first adjacency edge of v into Q;
10         if q ⊆ Vq and |Vq| ≥ L and k = 0 then
11             k ← the minimum weight among all popped edges; /* k
               is the connectivity of the SMCC_L */;
12 return Vq;
```

Implementation and Time Complexity. It is easy to see that Line 11 correctly sets k as the connectivity of the SMCC$_L$ of q, since any $(k+1)$-edge connected component of G either does not contain all vertices of q or has less than L vertices. Therefore, Algorithm 5 correctly computes the SMCC$_L$ of q by including all vertices reachable from v_0 through edges with weights at least k.

For the time complexity, it would be $O(|V_q| \log |V_q|)$ if we implement Q as a priority queue [12], since there are $|V_q|$ pop operations at Line 5, where V_q is the result. Nevertheless, we can achieve the time complexity of $O(|V_q|)$ by implementing Q using a bucket structure similar to that used in bin sort [12], in which each bucket $1 \leq b \leq k_{max}$ stores edges with weight b, where k_{max} is the maximum weight among all visited edges. Note that, V_q contains at least k_{max} vertices. Thus, the time complexity of Algorithm 5 is linear to the result size, and Algorithm 5 is optimal.

Example 4.3: Consider a query $q = \{v_1, v_4, v_5\}$ with $L = 6$, where the MST T is shown in Figure 3(a). Initially, $V_q = \{v_1\}$. After visiting the four adjacency edges of v_1, $V_q = \{v_1, \ldots, v_5\}$ with $|V_q| < 6$, thus, we further visit edge (v_4, v_7) to also include v_7 into V_q, and k is set as 3. Finally, V_q is obtained by also visiting $(v_5, v_9), (v_7, v_6), (v_6, v_8)$, and $V_q = \{v_1, \ldots, v_9\}$. □

4.6 Discussion: MST Selection

As discussed in Section 4.2, MST is not unique and any MST can be used to process our queries with proved correctness of query result. Regarding query processing time, the running time of SMCC-OPT (Algorithm 4) and SMCC$_L$-OPT (Algorithm 5) for processing SMCC and SMCC$_L$ queries is linear to the output size, which is independent to the selection of MST. Thus, the running time of SMCC-OPT and SMCC$_L$-OPT will not be affected by choosing a different MST; this also holds for the optimal algorithm (i.e., Algorithm 11) for processing steiner-connectivity queries, because the algorithm runs in linear time regarding the input size.

5. INDEX CONSTRUCTION AND MAINTENANCE

In this section, we propose efficient techniques for index construction (in Section 5.1) and for index maintenance (in Section 5.2) when the graph changes.

5.1 Index Construction

In the following, we present techniques for constructing the connectivity graph and the MST index, respectively.

5.1.1 Connectivity Graph Construction

A naive approach would be computing $sc(u, v)$ independently for each edge (u, v) in G by using the techniques in Section 3 (i.e., invoking Algorithm 1). Then, the time complexity would be at least $O(|V| \cdot h \cdot l \cdot |E|^2)$, since computing each $sc(u, v)$ by SMCC-BLE takes $O(|V| \cdot h \cdot l \cdot |E|)$ time.[1] Thus, this naive approach is time-consuming. We propose efficient algorithms below.

Batch Processing Algorithm. We can compute all $sc(\cdot, \cdot)$ values in a batch as follows. First, we initialize $sc(u, v)$ to be 1 for each edge (u, v) in G. Then, for each k varying from 2 to $|V|$, we compute k-edge connected components of G by invoking ComputeKECCs, and reassign $sc(u, v)$ to be k for each edge (u, v) in a k-edge connected component of G. The assigned $sc(\cdot, \cdot)$ values are correct steiner-connectivities when the algorithm terminates. The time complexity of this algorithm is $O(|V| \cdot h \cdot l \cdot |E|)$.

Computation Sharing Algorithm. In the above batch processing algorithm, we compute k-edge connected components of G by invoking ComputeKECCs independently for each k varying from 2 to $|V|$; however, a lot of computation can be shared among executions of ComputeKECCs for different k values. Therefore, we propose efficient computation sharing techniques below.

From the properties of k-edge connected components, we know that each k-edge connected component of G is a subgraph of a $(k-1)$-edge connected component of G. Therefore, all edges removed in computing $(k-1)$-edge connected components of G can also be safely removed when computing k-edge connected components of G; this means that, when computing k-edge connected components of G, we can take the set of $(k-1)$-edge connected components of G instead of G as the input. Moreover, instead of reassigning values of $sc(u, v)$ multiple times as done in the batch processing algorithm, we assign values of $sc(u, v)$ for each edge (u, v) only once when it is removed from G, based on the following lemma.

Lemma 5.1: *Given a $(k-1)$-edge connected graph G, $sc(u, v) = k-1$ for each edge (u, v) that is removed in computing k-edge connected components of G by* ComputeKECCs. □

The pseudocode of the computation sharing algorithm is shown in Algorithm 6. Initially, $\phi_1(G)$ consists only of G (Line 1). Then, we iteratively compute k-edge connected components of G by increasing k until there is no edge left (Lines 3–6). For a specific k, we use the set of subgraphs in $\phi_{k-1}(G)$ instead of G as input to ComputeKECCs (Line 5), and we assign $sc(u, v)$ to be $(k-1)$ for each edge (u, v) removed during computing k-edge connected components of G (Line 6). Note that, since the $|V|$-edge connected components of G contain no edge, Algorithm 6 shall terminate after at most $|V|$ iterations.

Algorithm 6: ConnGraph-Construction

Input: A graph G
Output: $sc(u, v)$ for each edge (u, v) in G
1 $\phi_1(G) \leftarrow \{G\}$; $k \leftarrow 1$;
2 **while** $\phi_k(G) \neq \emptyset$ **do**
3 $k \leftarrow k + 1$; $\phi_k(G) \leftarrow \emptyset$;
4 **for each** *graph g of size at least 2 in $\phi_{k-1}(G)$* **do**
5 $\phi_k(G) \leftarrow \phi_k(G) \cup$ ComputeKECCs(g, k); /* Compute k-edge connected components of g */
6 Assign $sc(u, v)$ to be $(k-1)$ for each edge (u, v) removed during the computation at Line 5;

Theorem 5.1: *Given a graph $G = (V, E)$, Algorithm 6 correctly computes $sc(u, v)$ for all edges $\{(u, v) \in E\}$ in $O(\alpha(G) \cdot h \cdot l \cdot |E|)$ total time, where $\alpha(G)$ is the* arboricity *of graph G [10].* □

[1]Note that, we discuss the complexities based on KECCs-Exact, while our following discussions also apply for KECCs-Random.

Note that, the arboricity $\alpha(G)$ of a graph G is the minimum number of forests into which G can be partitioned, and it is bounded by and usually much smaller than $\sqrt{|E|}$ [10]. It is interesting to see that this time complexity is even much smaller than the baseline algorithm (i.e., Algorithm 1) for computing the steiner-connectivity for one pair of vertices; nevertheless, Algorithm 6 is able to compute steiner-connectivities for $|E|$ pairs of vertices.

Example 5.1: Consider the graph in Figure 2. $\phi_2(G) = \{G\}$. $\phi_3(G) = \{g_1 \cup g_2, g_3\}$; (v_5, v_{12}) and (v_9, v_{11}) are removed, thus $sc(v_5, v_{12}) = 2$ and $sc(v_9, v_{11}) = 2$. In computing $\phi_4(G)$, all edges in G except those in g_1 are removed; therefore, these newly removed edges (u, v) have $sc(u, v) = 3$. Finally, the edges (u', v') in g_1 are removed and have $sc(u', v') = 4$. The connectivity graph is shown in Figure 3(a). □

5.1.2 MST Construction

Given the connectivity graph G_c of a graph G, the MST T can be constructed by a slight modification of the *minimum spanning tree* algorithms, such as Prim's algorithm or Kruskal's algorithm [12]. We omit the pseudocode here.

Implementation and Time Complexity. The MST T can be constructed in $O(|E|)$ time. We demonstrate how to achieve this time complexity by invoking Kruskal's algorithm to compute the minimum spanning tree of a graph. Firstly, edges in G_c can be sorted according to their weights in $O(|E|)$ time using bin sort [12], since they have integer weights in the range from 1 to $|V|$. Secondly, after sorting the edges, the time complexity of Kruskal's algorithm is $O(|E|\varphi(|V|))$ [12], where $\varphi(n)$ is the extremely slowly growing inverse of the single-valued Ackermann function and is less than 5 for all practical values of n [12]. Thus, given a connectivity graph G_c, the time complexity of constructing the MST T is $O(|E|)$. Note that, since the steiner-connectivities of edges are computed in sorted order in Algorithm 6, to compute MST we only need to incrementally maintain the (partial) MST in main memory by sequentially loading edges of the connectivity graph into main memory.

5.2 Index Maintenance

A graph is updated when a vertex or an edge is inserted or deleted. The insertion/deletion of an isolated vertex will not affect the connectivity graph, and inserting/deleting a vertex with associated edges can be regarded as inserting/deleting an isolated vertex followed by/following inserting/deleting edges; thus, we consider only the cases of edge insertion and edge deletion.

In the following, we first develop techniques to efficiently maintain the connectivity graph in Section 5.2.1 (for edge deletion) and in Section 5.2.2 (for edge insertion), and then propose techniques to efficiently update the MST in Section 5.2.3.

5.2.1 Handling Edge Deletion

When an edge (u, v) is deleted from G, we let $k_{u,v}$ denote $sc(u, v)$ in G before the deletion, let $g_{u,v}$ denote the $k_{u,v}$-edge connected component of G that contains u and v (i.e., the SMCC of $\{u, v\}$), and let G^- and $g^-_{u,v}$ denote the resulting graphs of G and $g_{u,v}$, respectively, by removing (u, v) from it.

We first present the lemma below to illustrate the maximum effect on the connectivity of a graph by deleting an edge from it.

Lemma 5.2: *Given a k-edge connected graph g, let g^- denote the resulting graph of deleting an edge $(u, v) \in g$ from g, then g^- is at least $(k-1)$-edge connected; g^- may be still k-edge connected.* □

For example, in Figure 2, g_1 is 4-edge connected and it becomes only 3-edge connected if we remove (v_2, v_3) from it; $g_1 \cup g_2$ is 3-edge connected and it is still 3-edge connected if we remove (v_2, v_3) from it. Now, in the lemma below we characterize the effect on the k-edge connected components of G when deleting an edge from G.

Lemma 5.3: *For any k-edge connected component g ($\neq g_{u,v}$) of G with $1 \leq k \leq |V|$, let g^- be the corresponding graph of g after removing (u, v), then g^- is also a k-edge connected component of G^-. Note that, g^- is the same as g if $(u, v) \notin g$.* □

Consider G in Figure 2, g_1, g_3, and $g_1 \cup g_2$ are 4-, 3-, and 3-edge connected component of G, respectively. After removing (v_2, v_3) from G, g_3^- and $(g_1 \cup g_2)^-$ are still 3-edge connected components of G, while g_1^- becomes only 3-edge connected since $g_1 = g_{v_2,v_3}$.

Observation-I. *From Lemma 5.3, it is immediate that for all edges in G^-, if the steiner-connectivity of an edge (u', v') changes due to the removal of (u, v), then (u', v') must be in $g_{u,v}^-$ and $sc(u', v') = k_{u,v}$ before the update; moreover, the updated $sc(u', v')$ is $k_{u,v} - 1$.*

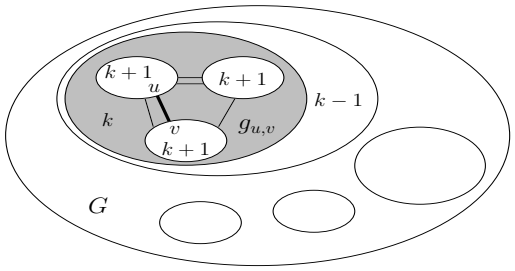


Figure 4: Illustration for connectivity graph maintenance

Conceptually, Figure 4 illustrates a graph G and its k-edge connected components for all different values of k, which are represented by ellipses; the connectivity values are shown inside the ellipses (e.g., $k-1, k, k+1$). From the properties of k-edge connected components, we know that each k-edge connected component is entirely contained in a $(k-1)$-edge connected component; thus, the ellipses are in a nested structure. When edge (u, v) (as indicated by the thick line) with $sc(u, v) = k$ is deleted from G, the SMCC of $\{u, v\}$, $g_{u,v}$, is indicated by the shadowed ellipse. Observation-I says that only edges in the shadowed area, which also excludes those in $(k+1)$-edge connected components, may have their steiner-connectivities changed, and they may change from k to $k-1$ due to decomposing $g_{u,v}^-$ into several k-edge connected components.

Algorithm. Based on Observation-I and the discussions above, we can update the connectivity graph by computing $k_{u,v}$-edge connected components of $g_{u,v}^-$ and assigning $sc(u', v')$ to be $k_{u,v} - 1$ for every edge (u', v') removed during the computation. Moreover, we can do better by contracting every $(k_{u,v} + 1)$-edge connected components of $g_{u,v}^-$ into a super-vertex, thus reduce the size of the graph to be processed. The pseudocode is shown in Algorithm 7.

Algorithm 7: ConnGraph-EdgeDeletion

Input: A connectivity graph G_c, and an edge (u, v)
Output: The updated connectivity graph G_c^-
1 Compute $k_{u,v}$ (= $sc(u, v)$), and $g_{u,v}$ (the SMCC of $\{u, v\}$); `/* By invoking Algorithm 4 */`;
2 Delete (u, v) from $g_{u,v}$ (obtaining $g_{u,v}^-$);
3 Contract every $(k_{u,v} + 1)$-edge connected components of $g_{u,v}^-$ into a super-vertex;
4 Compute $k_{u,v}$-edge connected components of $g_{u,v}^-$ and assign $sc(u', v')$ to be $k_{u,v} - 1$ for every edge removed during the computation;

Time Complexity. The time complexity of Algorithm 7 is $O(h \cdot l \cdot |g_{u,v}|)$ following from the time complexity of ComputeKECCs, where $|g_{u,v}|$ denotes the size of $g_{u,v}$. More specifically, Line 1 takes $O(|g_{u,v}|)$ time, Lines 2-3 take $O(|g_{u,v}|)$ time, while Line 4 takes $O(h \cdot l \cdot |g_{u,v}|)$ time. Therefore, updating the connectivity graph by Algorithm 7 is much more efficient than recomputing the connectivity graph by Algorithm 6 in Section 5.1.1 whose time complexity is $O(\alpha(G) \cdot h \cdot l \cdot |E|)$, since $|g_{u,v}| \ll |E|$ in practice.

Example 5.2: Suppose we delete edge (v_5, v_9) from the graph G in Figure 2, we first obtain $k_{v_5,v_9} = 3$ and $g_{v_5,v_9} = g_1 \cup g_2$. Then, we delete edge (v_5, v_9) from g_{v_5,v_9} and contract g_1 into a single super-vertex. Finally, we compute 3-edge connected components of the obtained graph during which we remove edges (v_4, v_7) and (v_5, v_7); thus, $sc(v_4, v_7) = sc(v_5, v_7) = 2$ for the new graph G^-. □

5.2.2 Handling Edge Insertion

When an edge (u, v) is inserted into G, we define $k_{u,v}$ and $g_{u,v}$ in the same way as for edge deletion; we let G^+ and $g_{u,v}^+$ denote the resulting graphs of G and $g_{u,v}$, respectively, after inserting (u, v).

Similar to Lemma 5.3, we have the following lemma to characterize the effect on the k-edge connected components of G when inserting an edge to G.

Lemma 5.4: *For any k-edge connected component g of G, let g^+ be the corresponding graph of g after inserting (u, v); note that, g^+ is the same as g if either u or v is not in g. Then, g^+ is also a k-edge connected component of G^+ if either $k \neq k_{u,v} + 1$ or $g^+ \nsubseteq g_{u,v}^+$.* □

Consider G in Figure 2, g_1, g_3, and $g_1 \cup g_2$ are 4-, 3-, and 3-edge connected component of G, respectively. After inserting (v_4, v_9) into G, g_1^+, g_3^+, and $(g_1 \cup g_2)^+$ are still 4-, 3-, and 3-edge connected component of G. However, if we instead insert (v_7, v_{10}) into G, then g_3^+ is no longer a 3-edge connected component of G, since $k = k_{v_7,v_{10}} + 1 = 3$ and $g_3^+ \subset g_{v_7,v_{10}} = G$; actually, $g_1 \cup g_2 \cup g_3$ (i.e., G) becomes the 3-edge connected component of G.

Observation-II. *From Lemma 5.4, it is immediate that for all edges in G^+, if the steiner-connectivity of an edge (u', v') changes due to the insertion of (u, v), then (u', v') must be in $g_{u,v}^+$ and $sc(u', v') = k_{u,v}$ before the update; moreover, the updated $sc(u', v')$ is $k_{u,v} + 1$.*

Reconsider Figure 4, assume that (u, v) is the edge we inserted into G. Observation-II says that only edges in the shadowed area, which also excludes those in $(k + 1)$-edge connected components, may have their steiner-connectivities changed, and they may change from k to $k + 1$ due to merging several $(k + 1)$-edge connected components of $g_{u,v}$ into a single $(k + 1)$-edge connected component.

Algorithm. Based on Observation-II and the discussions above, we can update the connectivity graph by computing $(k_{u,v} + 1)$-edge connected components of $g_{u,v}^+$ (by invoking ComputeKECCs), assigning $sc(u', v')$ to be $k_{u,v}$ for every edge removed during the computation, and assigning $sc(u'', v'')$ to be $k_{u,v} + 1$ for every other edge (u'', v'') in $g_{u,v}^+$ that has $sc(u'', v'') = k_{u,v}$ before the update. Moreover, we can do better by contracting every $(k_{u,v}+1)$-edge connected components of $g_{u,v}^+$ into a super-vertex, thus reduce the size of the graph to be processed. The pseudocode is shown in Algorithm 8.

Algorithm 8: ConnGraph-EdgeInsertion

Input: A connectivity graph G_c, and an edge (u, v)
Output: The updated connectivity graph G_c^+
1 Compute $k_{u,v}$ (= $sc(u, v)$), and $g_{u,v}$ (the SMCC of $\{u, v\}$); `/* By invoking Algorithm 4 */`;
2 Insert (u, v) into $g_{u,v}$ (obtaining $g_{u,v}^+$);
3 Contract every $(k_{u,v} + 1)$-edge connected components of $g_{u,v}^+$ into a super-vertex;
4 Assign all edges (u', v') in $g_{u,v}^+$ with $sc(u', v') = k_{u,v}$ to be $k_{u,v} + 1$;
5 Compute $(k_{u,v} + 1)$-edge connected components of $g_{u,v}^+$ and assign $sc(u', v')$ to be $k_{u,v}$ for every edge removed during the computation;

Time Complexity. Similar to the time complexity of Algorithm 7, the time complexity of Algorithm 8 is $O(h \cdot l \cdot |g_{u,v}|)$, where $|g_{u,v}|$ denotes the size of $g_{u,v}$.

Example 5.3: Suppose we insert edge (v_4, v_9) into the graph G in Figure 2. Firstly, we obtain $k_{v_4,v_9} = 3$ and $g_{v_4,v_9} = g_1 \cup g_2$. Then, we insert edge (v_4, v_9) into g_{v_4,v_9}, and contract g_1 into a single super-

vertex. Finally, we compute 4-edge connected components of the obtained graph during which we remove all the edges; thus, the connectivity graph remains the same except that we insert (v_4, v_9) into G_c with $sc(v_4, v_9) = 3$. □

5.2.3 MST Maintenance

Once the connectivity graph is updated, we also need to update the MST to reflect the changes of the connectivity graph. We summarise the updates on the connectivity graph in below.

- When edge (u, v) is deleted from G: (case-i), (u, v) is removed from G_c; and (case-ii), there may exist some edges (u', v') with $sc(u', v')$ changed from $k_{u,v}$ to $k_{u,v} - 1$.
- When edge (u, v) is inserted into G: (case-iii), (u, v) is inserted into G_c with $sc(u, v)$ to be $k_{u,v}$ or $k_{u,v} + 1$; and (case-iv), there may exist some edges (u', v') with $sc(u', v')$ changed from $k_{u,v}$ to $k_{u,v} + 1$.

Recall that MST is the maximum spanning tree of G_c. We let NT denote the set of non-tree edges of G_c (i.e., the set of edges of G_c that are not in MST). For example, the dotted edges in Figure 3(a) illustrate the NT corresponding to the MST in Figure 3(b). We organize all edges of NT into $|V|$ buckets (similar to the structure used in bin sort [12]), where each bucket $1 \le i \le |V|$ stores all edges of NT that have weight i and organizes these edges by a doubly linked list [12]. Based on this organization, we can locate the list of edges with a specific weight in constant time, and access edges of NT in non-increasing weight order in linear time.

In the following, we discuss how to update the MST for the above four cases separately, which also updates NT, based on the properties of maximum spanning tree.

Delete Edge (u, v) from G_c (case-i). 1) If (u, v) is in NT, then we just delete it from NT while MST remains the same. 2) Otherwise, we delete (u, v) from MST; this cuts MST into two trees. Then, we try to reconnect the two trees by an edge in NT with the maximum weight; that is, find the edge with the maximum weight such that its two end-vertices are in different trees. To efficiently find such an edge, we iterate over all edges (u'', v'') in NT in non-increasing weight order, terminate for the first one with u'' and v'' in different trees, and then move it from NT to MST. The pseudocode is shown in Algorithm 9.

Algorithm 9: MST-EdgeDeletion

```
   Input: A MST, a NT, and an edge (u, v)
   Output: The updated MST
1  if (u, v) is in NT then Remove (u, v) from NT;
2  else
3      Remove (u, v) from MST;
4      for each edge (u'', v'') in NT in non-increasing weight order do
5          if u'' and v'' are in two different trees of MST then
6              Move (u'', v'') from NT to MST;
7              break;
```

Implementation and Time Complexity. Note that, our organization of NT easily supports accessing edges of NT in non-increasing weight order. Thus, the most time-consuming part is Line 5 which tests whether two vertices are in the same tree of MST. To conduct the testing efficiently, we propose to represent each tree uniquely by its root vertex. Then, we identify the root vertices of the trees containing u'' and v'', respectively, and u'' and v'' are in the same tree if and only if the two root vertices are the same. Moreover, we propose to cache the results of finding root vertices for vertices, since we may need to conduct the testing for many edges.

The worst case time complexity would be $O(|NT| + |V|)$, where $|NT|$ denotes the number of edges in NT; the reason is that we may need to test Line 5 for all edges in NT, and the time complexity of finding the root vertices for all vertices is $O(|V|)$ time by using the cache optimization. In practice, however, we only need to test very few edges.

Decrement $sc(u', v')$ from $k_{u,v}$ to $k_{u,v} - 1$ for a Set of Edges (case-ii). 1) If (u', v') is in NT, then we just update the weight of (u', v') while MST remains the same. 2) Otherwise, we temporarily delete (u', v') from MST and try to reconnect the two resulting trees by an edge in NT with weight $k_{u,v}$ by using the same approach as above. If there exists such an edge (u'', v''), then we move (u', v') from MST to NT and move (u'', v'') from NT to MST. Otherwise, MST remains unchanged. Note that, here we can process all the updated edges in a batch.

Insert Edge (u, v) into G_c (case-iii). For this case, we compute the edge (u'', v'') in the path in MST between u and v that has the minimum weight. If $sc(u'', v'') < sc(u, v)$, then we replace (u'', v'') with (u, v) in MST and insert (u'', v'') into NT. Otherwise, we simply insert (u, v) into NT.

Increment $sc(u', v')$ from $k_{u,v}$ to $k_{u,v} + 1$ for a Set of Edges (case-iv). If (u', v') is in MST, then we just update the weight of (u', v'). Otherwise, similar to case-iii, we compute the edge (u'', v'') in the path in MST between u' and v' that has the minimum weight, and update MST and NT accordingly.

6. EXPERIMENTS

We conduct extensive performance studies to evaluate the efficiency of our index-based query processing techniques and our indexing techniques. Regarding SMCC queries, we evaluate the following algorithms:

- SMCC-BLE: the (exact) baseline algorithm in Section 3 by extending the existing exact technique in [7].
- SMCC-BLR: the (randomized) baseline algorithm in Section 3 by extending the existing randomized technique in [4].
- SMCC-OPT: the index-based optimal algorithm in Section 4.4 (Algorithm 4).

Regarding steiner-connectivity queries, we evaluate the following algorithms:

- SC-BL: the (exact) baseline algorithm in Section 3 by extending the existing technique in [7].
- SC-MST: the MST-based algorithm in Section 4.3 (Algorithm 3).
- SC-MST$*$: the MST*-based optimal algorithm in Section A.2 (Algorithm 11).

Regarding SMCC$_L$ queries, we evaluate the following algorithms:

- SMCC$_L$-BL: the (exact) baseline algorithm in Section 3 by extending the existing technique in [7].
- SMCC$_L$-OPT: the index-based optimal algorithm in Section 4.5 (Algorithm 5).

Regarding indexing, the following algorithms are evaluated:

- ConnGraph-B: the connectivity graph construction algorithm with batch processing in Section 5.1.1.
- ConnGraph-BS: the ConnGraph-B algorithm with further computation sharing in Section 5.1.1 (i.e., Algorithm 6).
- MST: the MST construction algorithm in Section 5.1.2.
- MST$*$: the MST* construction algorithm in Section A.2.
- IndexMaintain: the index maintenance algorithm in Section 5.2.

All algorithms are implemented in C++ and compiled with GNU GCC with the -O3 optimization; the source codes for computing k-edge connected components exactly (i.e., the KECCs-Exact algorithm) and randomly (i.e., the KECCs-Random algorithm) are obtained from the authors in [7] and [4], respectively. All experiments are conducted on a machine with an Intel(R) Xeon(R) 3.1GHz CPU and 128GB memory running Linux. We evaluate the performance of all algorithms on both real and synthetic graphs as follows.

Real Graphs: We evaluate the algorithms on eleven real graphs with their descriptions presented in Section A.4; sizes of these graphs are shown in Table 1, where the last column shows the average degree. We rank the graphs regarding the number of edges, and denote the graphs using IDs from $\{D1, \cdots, D11\}$. We regard the graphs with less than 1 million edges as *small graphs* (i.e., $\{D1, \cdots D4\}$), and others as *large graphs* (i.e., $\{D5, \cdots, D11\}$).

Type	ID	Dataset	#Edges	#Vertices	$\bar{d}$
small	D1	ca-GrQc	13,422	4,158	6.46
	D2	ca-CondMat	91,286	21,363	8.55
	D3	email-EuAll	339,925	224,832	3.02
	D4	soc-Epinions1	405,739	75,877	10.69
large	D5	amazon0601	2,443,311	403,364	12.11
	D6	web-Google	3,074,322	665,957	9.23
	D7	wiki-Talk	4,656,682	2,388,953	3.90
	D8	as-Skitter	11,094,209	1,694,616	13.09
	D9	LiveJournal	42,845,684	4,843,953	17.69
	D10	uk-2002	261,556,721	18,459,128	28.34
	D11	twitter-2010	1,202,513,344	41,652,230	57.7

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

Synthetic Graphs: We generate two kinds of synthetic graphs by the graph generator GTGraph[2], as follows.

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

We generate two power-law graphs, PL1 and PL2, with 12×10^4 and 14×10^4 edges, respectively, while both with 2×10^4 vertices. We also generate five SSCA graphs, SSCA1, SSCA2, SSCA3, SSCA4, and SSCA5, whose sizes are shown in Table 2. There are three small SSCA graphs and two large SSCA graphs.

Type	Dataset	#Edges	#Vertices	$\bar{d}$
small	SSCA1	24,584	4,096	12.00
	SSCA2	143,744	16,384	17.55
	SSCA3	896,759	65,536	27.37
large	SSCA4	5,640,272	262,144	43.03
	SSCA5	35,318,325	1,048,576	67.36

Table 2: Statistics of SSCA graphs ($\bar{d}$: average degree)

Queries. We studied SMCC, SMCC$_L$, and steiner-connectivity queries in this paper. For each kind of studied query, we generate five query sets, each of which consists of 1000 random queries of a particular size. The query size $|q|$ is chosen from $\{2, 5, 10, 20, 30\}$ with 10 as default. For each testing, we record the total running time of processing the 1000 queries in a query set. Each experiment is run three times, and the average CPU time is reported.

6.1 Evaluating Query Processing Techniques

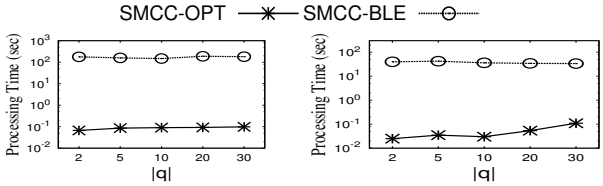
In this subsection, we evaluate the efficiency of our index-based optimal algorithm compared with baseline algorithms for each of the three queries we studied in this paper. Note that, the reported time is the total time of processing 1000 queries.

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

	Graph	SMCC-OPT	SMCC-BLE	SMCC-BLR
Real Graph	D1	0.001	2.66	851
	D2	0.15	28.7	18,302
	D3	0.09	148	-
	D4	0.26	256	-
Power-law	PL1	0.27	26	-
	PL2	0.26	36	-
SSCA	SSCA1	0.009	2.1	2,604
	SSCA2	0.03	36.3	35,447
	SSCA3	0.07	224	-

Table 3: Query processing time (SMCC query, in seconds)

Eval-I: SMCC Query. The running time of our optimal algorithm (i.e., SMCC-OPT) and two baseline algorithms (i.e., SMCC-BLE and SMCC-BLR) for processing SMCC queries on different graphs is shown in Table 3. In general, the running time of all three algorithms increases when the graph size increases. This is because both SMCC-BLE and SMCC-BLR traverse the entire graph several times, and the running time of SMCC-OPT is linear to the result size which generally becomes larger for random queries on larger graphs. Nevertheless, SMCC-OPT outperforms the two baseline algorithms by more than three orders of magnitude on all the graphs due to the optimality of SMCC-OPT. Moreover, the exact algorithm SMCC-BLE runs much faster than the randomized algorithm SMCC-BLR due to the large number of iterations in KECCs-Random which is invoked by SMCC-BLR; we set the number of iterations in KECCs-Random to be 50 which is used in [4]. Therefore, we exclude the randomized approach KECCs-Random from our further testings when computing k-edge connected components.



(a) D3 (email-EuAll) (b) SSCA2

Figure 5: Query processing time (SMCC query, vary $|q|$)

The running time of SMCC-OPT and SMCC-BLE on D3 (email-EuAll) and SSCA2 by varying the size $|q|$ of query is illustrated in Figure 5. We can see that, the running time of SMCC-OPT increases while that of SMCC-BLE remains almost the same, when $|q|$ increases. This is because SMCC-OPT runs linearly to the size of SMCC which generally becomes larger for random queries with more query vertices, while SMCC-BLE traverses the entire graph several times regardless the query. Nevertheless, SMCC-OPT outperforms SMCC-BLE by more than three orders of magnitude.

Graph	SMCC-OPT	Graph	SMCC-OPT
D5	13	D9	81
D6	6.1	D10	87
D7	2.9	D11	1.5
D8	18		
SSCA4	0.74	SSCA5	2.15

Table 4: Scalability testing for SMCC-OPT (in seconds)

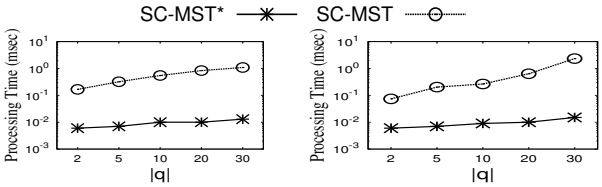
The scalability testing result of SMCC-OPT on large graphs is shown in Table 4, which shows that SMCC-OPT is very efficient for processing SMCC queries.

Eval-II: Steiner-Connectivity Query. The running time of SC-BL, SC-MST, SC-MST$*$ for processing steiner-connectivity queries on different graphs is shown in Table 5. Generally, the running time of SC-MST and SC-BL increases when the graph size increases, while that of SC-MST$*$ remains the same across different datasets. This is because SC-BL traverses the entire graph several times and

	Graph	SC-MST∗	SC-MST	SC-BL
Real Graph	D1	0.01	0.12	2,657
	D2	0.01	0.35	28,706
	D3	0.01	0.55	148,334
	D4	0.01	0.26	256,234
Power-law	PL1	0.01	0.26	26,275
	PL2	0.01	0.27	35,574
SSCA	SSCA1	0.01	0.16	2,095
	SSCA2	0.01	0.27	36,319
	SSCA3	0.01	0.66	224,170

Table 5: Query processing time (Steiner-connectivity query, in milliseconds)

SC-MST needs to first obtain the minimum connected subtree T_q of the MST T containing q, both of which increase as the graph size increases. However, the time complexity of SC-MST∗ is $O(|q|)$ which remains the same for different graphs (i.e., $|q| = 10$). Overall, SC-MST runs up-to six orders of magnitude faster than SC-BL, and SC-MST∗ runs more than one order of magnitude faster than SC-MST; this confirms the efficiency of SC-MST∗.



(a) D3 (email-EuAll) (b) SSCA2

Figure 6: Query processing time (Steiner-connectivity query, vary $|q|$)

The running time of SC-MST∗ and SC-MST on D3 (email-EuAll) and SSCA2 by varying the query size $|q|$ is demonstrated in Figure 6. We can see that both SC-MST∗ and SC-MST run slower when $|q|$ increases; this is due to their time complexities (i.e., $O(|q|)$ and $O(|T_q|)$, respectively) which become larger for larger $|q|$. Nevertheless, the running time of SC-MST∗ increases much slower than that of SC-MST, and SC-MST∗ outperforms SC-MST by up-to two orders of magnitude.

The scalability of SC-MST and SC-MST∗ on large graphs is confirmed in Table 10 in the Appendix.

	Graph	SMCC$_L$-OPT	SMCC$_L$-BL
Real Graph	D1	0.01	2.65
	D2	0.12	26
	D3	0.07	158
	D4	0.22	242
Power-law	PL1	0.24	22
	PL2	0.25	31
SSCA	SSCA1	0.01	2.06
	SSCA2	0.04	25.3
	SSCA3	0.15	250

Table 6: Query processing time (SMCC$_L$ query, in seconds)

Eval-III: SMCC$_L$ Query. The running time of SMCC$_L$-OPT and SMCC$_L$-BL for processing SMCC$_L$ queries on different graphs is shown in Table 6, which has similar trends to that of SMCC-OPT and SMCC-BLE in Table 3. The optimal algorithm SMCC$_L$-OPT outperforms the baseline algorithm SMCC$_L$-BL by more than three orders of magnitude. The scalability of SMCC$_L$-OPT on large graphs is confirmed in Table 11 in the Appendix.

6.2 Evaluating Indexing Techniques

In this subsection, we evaluate the efficiency of our indexing algorithms, ConnGraph-B, ConnGraph-BS, MST, and MST∗, and our index maintenance algorithm, IndexMaintain. Note that, the reported time does not include the time to load in the input graph from disk to main memory, since it is the same for the compared algorithms and it is small compared with the computation time.

Graph	ConnGraph-B	ConnGraph-BS	MST	MST∗
D1	0.054	0.019	0.001	0.003
D2	0.3	0.154	0.005	0.005
D3	2.3	0.332	0.049	0.036
D4	10.12	3.38	0.064	0.013
D5	26	23	0.468	0.083
D6	82.8	27.7	0.626	0.159
D7	202	44	1.2	0.482
D8	511	141	1.86	0.33
D9	7,766	1,450	9.17	1.425
D10	33,143 (9.2hrs)	6,172 (1.7hrs)	21	3.429
D11	-	61hrs	151	7.8
PL1	0.211	0.171	0.006	0.004
PL2	0.3	0.268	0.007	0.004
SSCA1	0.072	0.041	0.001	0.003
SSCA2	0.867	0.5	0.008	0.004
SSCA3	16.86	6.66	0.112	0.01
SSCA4	264	70.57	0.796	0.05
SSCA5	2,289	720	6.78	0.25

Table 7: Indexing time (in seconds)

Eval-IV: Indexing Time. The running time of ConnGraph-B and ConnGraph-BS for connectivity graph construction is presented in the second and third columns of Table 7, respectively. We can see that ConnGraph-BS is on average more than three times faster than ConnGraph-B; for example, the running time of ConnGraph-BS and ConnGraph-B on D10 (uk-2002) is 1.7hrs and 9.2hrs, respectively. This is because ConnGraph-BS improves upon ConnGraph-B by further utilizing the computation sharing technique; that is, ConnGraph-BS takes the output of the previous iteration as the input of the current iteration (thus, the input size gradually reduces).

The running time of MST and MST∗ (for constructing MST and MST*, respectively) is shown in the fourth and fifth columns of Table 7, respectively. We can see that their running time conforms with their time complexities (i.e., $O(|E|)$ for MST and $O(|V|)$ for MST∗. Note that, MST takes a connectivity graph as input while MST∗ takes an MST as input. From Table 7, we can see that the index structures can be constructed very efficiently.

Graph	|MST|	$\|G_c\|$	Graph	|MST|	$\|G_c\|$
D1	0.14M	0.15M	D10	649M	3.0G
D2	0.75M	1.1M	D11	1.3G	14G
D3	7.9M	3.9M	PL1	0.57M	1.4M
D4	2.6M	4.7M	PL2	0.57M	1.6M
D5	14M	28M	SSCA1	0.14M	0.28M
D6	23M	36M	SSCA2	0.57M	1.7M
D7	84M	54M	SSCA3	2.3M	11M
D8	59M	127M	SSCA4	9.2M	65M
D9	170M	491M	SSCA5	37M	405M

Table 8: Index size (in Bytes)

Eval-V: Index Size. The sizes of the MST index for the tested graphs are demonstrated in Table 8, which conform with the theoretical analysis (i.e., $O(|V|)$). For example, the MST index size of the SSCA graphs increases by about 4 times when the number of vertices increases by 4 times (i.e., from SSCA1 to SSCA5). Overall, the MST index size is small; for example, it is 1.3G for the largest graph, D11 (twitter), which has 1.2 billion edges. Thus, we can store the index in main memory.

The sizes of the connectivity graphs for the tested graphs are also shown in Table 8. Note that, a connectivity graph is a weighted version of the input graph; that is, it consists of the input graph and weights of its edges. Thus, the reported size $|G_c|$ in Table 8 also includes the input graph size. In general, the connectivity graph size $|G_c|$ is larger than the size of MST because there are more edges

than vertices in a graph. However, there is an exception on D3 (email-EuAll) and D7 (wiki-Talk), which have small average degrees ($\bar{d}$); because in MST we store the level, the edge to its parent, and the weight of the edge for each vertex.

Graph	IndexMaintain	Graph	IndexMaintain
D1	0.226	D10	3,130
D2	0.054	PL1	36.9
D3	3.45	PL2	35.7
D4	24.5	SSCA1	0.068
D5	906	SSCA2	0.37
D6	1.98	SSCA3	4.59
D7	82	SSCA4	10.7
D8	9.58	SSCA5	35.2
D9	48.9		

Table 9: Average index updating time (in milliseconds)

Eval-VI: Index Maintenance. The index updating time for the tested graphs is shown in Table 9. The reported time is the average running time for updating the index structure for a sequence of 40 mix-up updating operations which consists of 20 edge deletions and 20 edge insertions. Note that, the average updating time for edge insertion is similar to that for edge deletion. By comparing Table 9 with Table 7, we can see that our index maintenance techniques can maintain the index structure very efficiently; for example, the running time of IndexMaintain is more than three orders of magnitude smaller than that of computing the index from scratch. Note that, the time in Table 9 is in milliseconds, while that in Table 7 is in seconds. Moreover, by comparing Table 9 with Table 3 (specifically, the running time of SMCC-BLE), we can see that the strategy of updating the index structure first and then processing queries works; that is, the total time of updating the index and then using it to process queries is much smaller than the running time of the SMCC-BLE algorithm which processes queries directly on G without any index.

7. EXTENSIONS

Our techniques can be extended to process other queries and to conduct external-memory computation as well.

Subset-SMCC Query. Given a set q of query vertices and a number L ($\leq |q|$), a subset-SMCC query is to find the maximum induced subgraph with the maximum connectivity such that it contains at least L vertices in q.

Subset-SMCC queries can be processed by extending our prioritized search technique in Section 4.5 as follows. We conduct $|q|$ prioritized searches, one starting from each vertex $v \in q$. For each prioritized search, we set the connectivity k of the component once V_q includes at least L vertices of q; that is, at Line 10 of Algorithm 5, we check whether V_q contains at least L vertices in q. Then, we choose the component with maximum connectivity among the $|q|$ computed components to be the result of the query.

SMCC-Cover Query. Given a set q of query vertices and a number L ($\leq |q|$), the SMCC-cover query is to find L maximum induced subgraphs such that they collectively cover q (i.e., contain q) and the minimum of the connectivities of the L subgraphs is maximized.

SMCC-cover queries can also be processed by extending our prioritized search technique in Section 4.5. Similar to processing subset-SMCC queries in the above, we also instantiate $|q|$ instances of prioritized search, denoted $PS_0, \ldots, PS_{|q|-1}$, one for each vertex $v_i \in q$ with $0 \leq i \leq |q| - 1$. However, different from processing subset-SMCC queries in which the $|q|$ prioritized searches are executed independently, we run the $|q|$ prioritized searches in a coordinated manner. Each instance PS_i has a weight which is defined as the minimum weight of the edges popped from Q for that instance (see Line 11 of Algorithm 5). Each time, we only run one of the $|q|$ prioritized searches by popping one edge from Q and updating Q and V_q; the instance is chosen as the one that has the maximum weight. Once an instance PS_i visits a vertex that has already been visited by another instance PS_j, we merge these two instances into a single instance. The algorithm terminates once there are exactly L instances left, and each remaining instance sets the corresponding k and returns the component.

Steiner-Connectivity with Size Constraint. Similar to the definition of steiner-connectivity which is the connectivity of the SMCC, the steiner-connectivity with size constraint is the connectivity of SMCC_L, denoted gc_l. Given a set q of query vertices and a number L, this query can be directly processed by Algorithm 5; that is, we return the value of k instead of the SMCC_L.

External-Memory Computation. Below, we discuss possible structures of disk-based index and possible ways to conduct external-memory computation.

External-Memory Query Processing. We can store the MST T on disk in the form of adjacency lists which are stored in consecutive blocks, and store the starting position of the adjacency list for each vertex using a B+ tree. During query processing, whenever we need to access the adjacency list of a particular vertex, we load the corresponding blocks into main memory if they have not already been loaded in.

External-Memory Index Construction and Maintenance. The input graph can be organized on disk in a similar way to MST discussed above. Regarding computing the connectivity graph, we can further utilize the sparsification techniques in [23] and the contraction idea (i.e., contracting k-edge connected components into super-vertices when computing k'-edge connected components for all $k' < k$) in Sections 5.2.1 and 5.2.2 to reduce the number of vertices and/or edges to be loaded into main memory. Efficient external-memory algorithms for computing the minimum spanning tree have been studied in [15]. Regarding index maintenance, for deleting/inserting an edge (u, v), we only need to load the SMCC of $\{u, v\}$ into main memory while also contracting k-edge connected components in the SMCC with $k = sc(u, v) + 1$ into super-vertices.

8. CONCLUSION

In this paper, we studied the SMCC query and its variants, and proposed novel index-based optimal algorithms. For processing SMCC queries, we proposed an efficient algorithm based on a novel index such that the algorithm runs in linear time regarding the result size (thus, optimal). We also proposed an efficient algorithm to process SMCC_L queries in linear time regarding the result size. For steiner-connectivity queries, we proposed an optimal algorithm to run in $O(|q|)$ time, which is linear to the query size (thus, optimal). To build the index, we extended the techniques in [7] to accommodate batch processing and computation sharing. To efficiently support the applications with graph updates, we also presented novel incremental techniques. We finally conducted experiments to evaluate the efficiency of our approaches, and the experimental results demonstrated that our index-based algorithms outperform baseline algorithms by several orders of magnitude and our index maintenance techniques can update the index structure very efficiently when graph changes.

Acknowledgements. Lijun Chang is supported by ARC DE150100563. Xuemin Lin is supported by NSFC61232006, ARC DP120104168, ARC DP140103578, and ARC DP150102728. Lu Qin is supported by ARC DE140100999. Jeffrey Xu Yu is supported by Research Grants Council of the Hong Kong SAR, China, 14209314 and 418512. Wenjie Zhang is supported by ARC DE120102144, DP120104168, ARC DP150103071 and DP150102728.

9. REFERENCES

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

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

[3] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. On finding lowest common ancestors in trees. In *Proc. of STOC'73*, 1973.

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

[5] M. A. Bender and M. Farach-Colton. The lca problem revisited. In *Proc. of LATIN'00*, 2000.

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

[7] 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. SIGMOD'13*, 2013.

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

[9] 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.

[10] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM J. Comput.*, 14(1), 1985.

[11] A. Clauset. Finding local community structure in networks. *Phys. Rev. E*, 72, 2005.

[12] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[13] W. Cui, Y. Xiao, H. Wang, Y. Lu, and W. Wang. Online search of overlapping communities. In *Proc. of SIGMOD'13*, 2013.

[14] W. Cui, Y. Xiao, H. Wang, and W. Wang. Local search of communities in large graphs. In *Proc. of SIGMOD'14*, 2014.

[15] R. Dementiev, P. Sanders, D. Schultes, and J. F. Sibeyn. Engineering an external memory minimum spanning tree algorithm. In *IFIP TCS*, 2004.

[16] C. Dorn and S. Dustdar. Composing near-optimal expert teams: A trade-off between skills and connectivity. In *Proc. of OTM'10*, 2010.

[17] A. Gibbons. *Algorithmic Graph Theory*. Cambridge University Press, 1985.

[18] R. E. Gomory and T. C. Hu. Multi-Terminal Network Flows. *Journal of the Society for Industrial and Applied Mathematics*, 9(4), 1961.

[19] E. Hartuv and R. Shamir. A clustering algorithm based on graph connectivity. *Information Processing Letters*, 76, 1999.

[20] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu. Querying k-truss community in large and dynamic graphs. In *Proc. of SIGMOD'14*, 2014.

[21] J. A. Kelner, Y. T. Lee, L. Orecchia, and A. Sidford. An almost-linear-time algorithm for approximate max flow in undirected graphs, and its multicommodity generalizations. In *Proc. of SODA'13*, 2013.

[22] T. Lappas, K. Liu, and E. Terzi. Finding a team of experts in social networks. In *Proc. of KDD'09*, 2009.

[23] H. Nagamochi and T. Ibaraki. A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica*, 7(5&6), 1992.

[24] J. B. Orlin. Max flows in o(nm) time, or better. In *Proc. of STOC'13*, 2013.

[25] 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.

[26] J. Sherman. Nearly maximum flows in nearly linear time. In *Proc. of FOCS'13*, 2013.

[27] M. Sozio and A. Gionis. The community-search problem and how to plan a successful cocktail party. In *Proc. of KDD'10*, 2010.

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

[29] 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.

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

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

[32] 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.

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

[34] 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.

A. APPENDIX

A.1 Pseudocode for Steiner-Connectivity Query

The pseudocode for processing a steiner-connectivity query in $O(|T_q|)$ time is shown in Algorithm 10 blow.

Algorithm 10: SC-MST

```
Input: A MST T, and a set q = {v_0, ..., v_{|q|-1}} of vertices
Output: The steiner-connectivity of q
T_q ← ∅;
for i ← 1 to |q| − 1 do
    if i = 1 then u ← v_0 else u ← lca_{i−1};
    v ← v_i;
    while u ≠ v do
        if l(u) ≥ l(v) then u ← p(u);
        else
            v ← p(v);
            if v has already been visited then break;
        Add the visited edge to T_q;
        Mark u and v as visited;
    lca_i ← u;
return sc(q) = min_{(u,v)∈T_q} λ_T(u, v);
```

A.2 MST*: An Optimal Algorithm

In Algorithm 3, we process a steiner-connectivity query q in $O(|T_q|)$ time by first computing the subtree T_q. To improve the query efficiency, we propose to construct a new index structure from the MST T by reorganizing edges such that we can compute the steiner-connectivity of q without computing T_q.

MST*. Given a MST T, we propose to further process it to generate an *optimization connectivity preserving index (MST*)*, denoted T^*. A MST* T^* is a rooted tree consisting of two types of vertices: 1) *vertex-type vertex*, each such vertex corresponds to one vertex in T; and 2) *edge-type vertex*, each such vertex corresponds to one edge in T. Each edge-type vertex u has a weight, denoted as $\lambda_T(u)$, which is the weight of the corresponding edge in T. In T^*, vertex-type vertices are leaf vertices and edge-type vertices are intermediate vertices. In the following, we let $e_{u,v}$ denote the edge-type vertex in T^* that corresponds to edge (u, v) in T.

The edges of T^* are defined (or constructed) recursively. Let (u, v) be the edge with minimum weight in T. By removing (u, v) from T, we get two connected trees T_1 and T_2 which are subtrees of T. The MST* of T_1 and T_2 are defined (or constructed) recursively; let T_1^* and T_2^* be the corresponding MST* constructed for T_1 and T_2, respectively. Then, T^* is the tree constructed by adding an edge from $e_{u,v}$ to the root of T_1^* and an edge from $e_{u,v}$ to the root of T_2^*; here, $e_{u,v}$ is the edge-type vertex corresponding to (u, v). The base case is that, if T consists of a single vertex, then the corresponding MST* T^* is the same as T consisting of a single vertex-type vertex.

Figure 7 shows the corresponding MST* T^* constructed from the MST T in Figure 3(b), where $e_{i,j}$ corresponds to edge (v_i, v_j) in

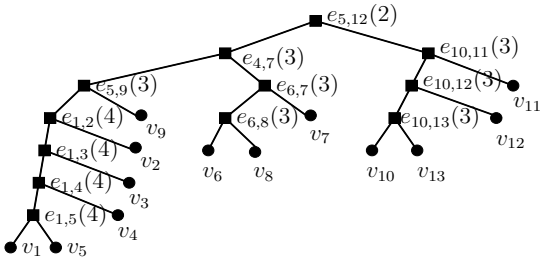


Figure 7: MST*: a rooted tree

T and weights are shown in brackets. $e_{5,12}$ is the root of T^*, since (v_5, v_{12}) is the edge with minimum weight in T.

Properties of MST*. By reorganizing edges of MST T into MST* T^*, T^* has the following properties, which can be utilized to efficiently process steiner-connectivity queries.

Lemma A.1: *A MST* T^* is a complete binary tree [17]; that is, each vertex has either two children or no children. For any leaf-to-root path in T^*, the weights of edge-type vertices in it are in non-increasing order.* □

Proof Sketch: Firstly, it is easy to see that each edge-type vertex has two children, one corresponding to each connected subtree after removing the corresponding edge in a MST. Thus MST* T^* is a complete binary tree. Secondly, it is obvious that for each subtree in T^*, the root (edge-type) vertex has the minimum weight. Thus, all descendant edge-type vertices have weights no less than its own weight. Thus, the lemma holds. □

Lemma A.2: *For any two vertex-type vertices (i.e., leaf vertices) u and v in T^*, let e be the LCA (see Definition 4.3) of u and v in T^*, then $sc(u, v)$ is equal to the weight of e.* □

Proof Sketch: From Lemma 4.4, let the undirected path between u and v in T be $P = (v_1(= u), v_2, \ldots, v_n(= v))$, then we have $sc(u, v) = \min_{i=1}^{n-1} \lambda_T(v_i, v_{i+1})$. Let T_e^* be the subtree of T^* rooted at e, and T_e be the subtree of T corresponding to T_e^*. Then, all edges in P are in T_e since u and v are still connected in T_e, and P contains e since u and v are disconnected by removing e from T_e. Therefore, the weight of e is equal to $\min_{i=1}^{n-1} \lambda_T(v_i, v_{i+1})$, since e has the minimum weight among all edges in T_e. Thus, the lemma holds. □

Processing Steiner-Connectivity Queries. Following from Lemma A.2 and Lemma 4.2, given a MST* T^*, we can process a steiner-connectivity query $q = \{v_0, \ldots, v_{|q|-1}\}$ by first computing $sc(v_0, v_i)$, $\forall 1 \le i \le |q| - 1$, which is the weight of the LCA of v_0 and v_i in T^*, and then returning the minimum value as $sc(q)$. The pseudocode is shown in Algorithm 11.

Algorithm 11: SC-OPT

```
Input: A MST* T*, and a set q = {v_0, ..., v_{|q|-1}} of query vertices
Output: The steiner-connectivity of q
1 sc(q) ← +∞;
2 for i ← 1 to |q| − 1 do
3     e ← LCA(v_0, v_i);
4     if λ_T(e) < sc(q) then sc(q) ← λ_T(e);
5 return sc(q);
```

Implementation and Time Complexity. Algorithm 11 can be implemented to process a steiner-connectivity query in $O(|q|)$ time, since the LCA of two vertices in a rooted tree at Line 3 of Algorithm 11 can be identified in constant time by existing techniques with linear preprocessing time and space (Please refer to [5] for details).

Example 1.1: Consider the MST* T^* in Figure 7. To compute $sc(v_8, v_{13})$, we first identify the LCA of v_8 and v_{13} which is $e_{5,12}$ with weight 2; therefore, $sc(v_8, v_{13}) = 2$. Now, consider another vertex pair v_8 and v_7. The LCA of v_8 and v_7 is $e_{6,7}$ with weight 3. Thus, $sc(\{v_8, v_{13}, v_7\}) = \min\{sc(v_8, v_{13}), sc(v_8, v_7)\} = 2$. □

MST* Construction. Constructing T^* in a top-down fashion by following the definition described above requires quadratic time, since each time after removing an edge from T we need to recompute to obtain the edge with minimum weight in each of the resulting subtrees which requires linear time. Therefore, we propose an algorithm to construct T^* in linear time in a bottom-up fashion.

The algorithm is shown in Algorithm 12, which works similar to Kruskal's algorithm for computing minimum spanning tree [12]; that is, we merge small trees into bigger trees by adding edge-type vertices and edges from the newly added vertices to roots of existing trees, and finally get T^*. Nevertheless, this is nontrivial. Initially, each vertex in T forms a rooted tree (Line 1). We sort all edges E_T of T in a non-increasing weight order (Line 2). For each edge (v_{i_1}, v_{i_2}) in E_T in the sorted order, we perform the following three operations: 1) add an edge-type vertex e_{i_1,i_2} to T^* (Line 4); 2) identify the current roots of the trees containing v_{i_1} and containing v_{i_2}, respectively, by FindRoot(v_{i_1}) and FindRoot(v_{i_2}); 3) add edges from e_{i_1,i_2} to each of the two identified root vertices (Lines 5–6). The implementations of sorting edges at Line 2 and FindRoot at Lines 5–6 shall be discussed shortly.

Algorithm 12: MST*-Construction

```
Input: A MST T
Output: A MST* T*
1 T* is initialized to contain |V| isolated vertices of T; /* each vertex
  forms a rooted tree */;
2 Sort all edges E_T of T in non-increasing weight order;
3 for each edge (v_{i_1}, v_{i_2}) in E_T in sorted order do
4     Add a new vertex e_{i_1,i_2} to T*, with weight λ_T(v_{i_1}, v_{i_2});
5     Add an edge from e_{i_1,i_2} to the root of the tree containing v_{i_1}
      (found by invoking FindRoot(v_{i_1})), to T*;
6     Add an edge from e_{i_1,i_2} to the root of the tree containing v_{i_2}
      (found by invoking FindRoot(v_{i_2})), to T*;
7 return T*;
```

Theorem 1.1: *Algorithm 12 correctly constructs MST* T^* from MST T.* □

Proof Sketch: Without loss of generality, we assume that all edge weights in T are different. For an edge (v_{i_1}, v_{i_2}) in T, we let $T^{(v_{i_1}, v_{i_2})}$ denote the connected subtree of T containing (v_{i_1}, v_{i_2}) by removing all edges with weights less than $\lambda_T(v_{i_1}, v_{i_2})$ from T. Then $T^{(u,v)} = T$ when (u, v) is the edge with minimum weight in T. We prove by induction that after processing (v_{i_1}, v_{i_2}) at Line 3, the current subtree rooted at e_{i_1,i_2} is the corresponding MST* of $T^{(v_{i_1}, v_{i_2})}$.

For the first edge processed at Line 3, the above claim is obviously true. Now, consider an arbitrary edge (v_{i_1}, v_{i_2}) processed at Line 3. Without loss of generality, assume that e_{j_1,j_2} and e_{k_1,k_2} are the two children of e_{i_1,i_2} in the MST* of $T^{(v_{i_1}, v_{i_2})}$, and e_{j_1,j_2} and e_{k_1,k_2} have v_{i_1} and v_{i_2} as their descendants, respectively. Then, by induction, the MST* of $T^{(v_{j_1}, v_{j_2})}$ and that of $T^{(v_{k_1}, v_{k_2})}$ must have been constructed, since the weights of (v_{j_1}, v_{j_2}) and (v_{k_1}, v_{k_2}) are larger than that of (v_{i_1}, v_{i_2}). Thus, FindRoot(v_{i_1}) will return e_{j_1,j_2}, and FindRoot(v_{i_2}) will return e_{k_1,k_2}. Therefore, the theorem holds. □

Implementation and Time Complexity. Algorithm 12 can be implemented in $O(|V|)$ time as follows. Firstly, sorting all edges of T at Line 2 can be conducted in $O(|V|)$ time using bin sort [12], since the weights of all $(|V| - 1)$ edges in T are integers in the range from 1 to $|V|$. Secondly, the time-critical component of Algorithm 12 (i.e., FindRoot at Lines 5–6) can be implemented by the union-find algorithm on a disjoint-set data structure [12]. Note that, to achieve linear time for the union-find algorithm on a disjoint-set data struc-

ture, two optimization techniques need to be applied: *union by rank* and *path compression*. However, at a first glance, the union by rank optimization is not applicable here, because we need to point the parents of the root vertices obtained at Lines 5–6 to e_{i_1,i_2} deterministically. Thus, by the path compression optimization alone, the total running time of all FindRoots in Algorithm 12 would be $O(|V| + |V| \log |V|)$ [12].

Nevertheless, we can modify the union-find algorithm and the disjoint-set data structure to enable both union by rank and path compression optimizations, and thus achieve $O(|V|)$ total running time, as follows. Note that, here we have two forests: one is T^* and the other is the disjoint-set data structure; there is a one-to-one correspondence between trees in T^* and trees in the disjoint-set data structure (i.e., each pair of corresponding trees contain the same set of vertices). For each root of a tree in the disjoint-set data structure, we also store a pointer pointing to the actual root vertex of the corresponding tree in T^*. At Lines 5–6, we need to update both T^* and the disjoint-set data structure. T^* is updated as stated at Lines 5–6. For updating the disjoint-set data structure, we union the following three trees using the union by rank optimization: one consisting of e_{i_1,i_2}, one containing v_{i_1}, and the other containing v_{i_2}; we also point the pointer of the root of the resulting tree in the disjoint-set data structure to e_{i_1,i_2} which is in T^*. Thus, FindRoot(v) returns the vertex in T^* that is pointed by the root of the tree containing v in the disjoint-set data structure.

Thus, the $2(|V| - 1)$ FindRoot operations in Algorithm 12 can be conducted in $O(|V|\varphi(|V|))$ total time, where $\varphi(n)$ is less than 5 for all practical values of n [12]. Consequently, Algorithm 12 can be implemented in $O(|V|)$ time. We omit the pseudocode.

Example 1.2: For example, consider the MST T in Figure 3(b), assume the edges E_T of T are sorted as $\{(v_1, v_5), (v_1, v_4), (v_1, v_3), (v_1, v_2), (v_{10}, v_{13}), (v_{10}, v_{12}), (v_{10}, v_{11}), (v_6, v_8), (v_6, v_7), (v_5, v_9), (v_4, v_7), (v_5, v_{12})\}$. After processing the first four edges (i.e., till (v_1, v_2)), we get the subtree rooted at $e_{1,2}$ of the tree in Figure 7. After processing the first seven edges (i.e., till (v_{10}, v_{11})) we additionally get the subtree rooted at $e_{10,11}$. After processing all twelve edges except the last one, we get two subtrees, one rooted at $e_{4,7}$ and the other rooted at $e_{10,11}$. After processing (v_5, v_{12}), we get the final tree shown in Figure 7, which is the MST* T^*. □

A.3 Proofs of Lemmas and Theorems

Proof Sketch of Lemma 4.1: Let V_q be the set of vertices such that $sc(\{v_0, v\}) \geq sc(q)$, $\forall v \in V_q$; that is, $V_q = \{v \in V \mid sc(\{v_0, v\}) \geq sc(q)\}$. We prove that V_q is the SMCC of q.

Firstly, it is obvious that every vertex v in the SMCC of q must satisfy $sc(\{v_0, v\}) \geq sc(q)$, since the SMCC of q is $sc(q)$-edge connected. Thus, V_q is a super-set of the SMCC of q. Secondly, for each vertex v satisfying $sc(\{v_0, v\}) \geq sc(q)$, it must be included in the SMCC of q, since the SMCC of q is the $sc(q)$-edge connected component of G which is maximal. Thus, the lemma holds. □

Proof Sketch of Lemma 4.2: Firstly, we can prove that $sc(q) \leq sc(v_0, v_i)$ for $1 \leq i \leq |q| - 1$. This is because any k-edge connected component of G is entirely contained in a $(k-1)$-edge connected component of G, and the SMCC of q is the k-edge connected component of G with the maximum k that contains q. Thus, $sc(q) \leq \min_{1 \leq i \leq |q|-1} sc(v_0, v_i)$.

Secondly, we can prove that $sc(q) \geq \min_{1 \leq i \leq |q|-1} sc(v_0, v_i)$. Without loss of generality, assume $sc(v_0, v_1)$ has the smallest value among $\{sc(v_0, v_i) \mid 1 \leq i \leq |q| - 1\}$, and let g be the SMCC of $\{v_0, v_1\}$. Then, for any v_i, since $sc(v_0, v_i) \geq sc(v_0, v_1)$, g also contains v_i following from the properties of k-edge connected components. Thus, g contains all vertices of q, and $sc(q) \geq sc(v_0, v_1)$.

Therefore, the lemma holds. □

Proof Sketch of Lemma 4.3: Let P_m be the path in $\mathcal{P}_{u,v}$ with the maximum weight (i.e., $P_m = \arg\max_{P \in \mathcal{P}_{u,v}} w(P)$). We first prove that $sc(u, v) \geq w(P_m)$. Let $P_m = (v_1(= u), \ldots, v_i, v_{i+1}, \ldots, v_n(= v))$. Without loss of generality, assume that $w(P_m) = w(v_i, v_{i+1})$, and let k be this value. Then, there is a k-edge connected component g of G containing v_i and v_{i+1}. Based on the properties of k-edge connected components and the fact that $sc(v_{i-1}, v_i) \geq sc(v_i, v_{i+1}) = k$, g should also contain v_{i-1}. Similarly, we can prove that g contains all vertices in P_m, which include u and v. Therefore, $sc(u, v) \geq k = w(P_m)$.

Now we prove that $sc(u, v) \leq w(P_m)$ by contradiction. Assume that $sc(u, v) > w(P_m)$, and let k be $sc(u, v)$. Then, there is a k-edge connected component g of G containing u and v, and the set of edges in G_c that correspond to edges in g have weights at least k. Since g is connected, we can find a path between u and v in g with minimum weight at least k ($> w(P_m)$), which contradicts that P_m is the path with maximum weight among all paths between u and v (i.e., $\mathcal{P}_{u,v}$). Therefore, $sc(u, v) \leq w(P_m)$. Thus, the lemma holds. □

Proof Sketch of Lemma 4.4: Note that, $\lambda_T(v_i, v_{i+1}) = w(v_i, v_{i+1})$ for any edge (v_i, v_{i+1}) in T. We prove the lemma by contradiction. Assume there is a path P' in G_c between u and v with a weight larger than P (i.e., $w(P') > w(P)$). Obviously, not all edges of P' are in T. Without loss of generality, assume that $(v_1, v_2) \in P$ has the minimum weight among all edges in P, then $(v_1, v_2) \notin P'$ since $w(P') > w(P)$. Therefore, there is a simple cycle in $P \cup P'$ such that (v_1, v_2) is in the cycle and has the minimum weight. According to the cycle property of maximum spanning tree, (v_1, v_2) cannot be in T. Contradiction. Thus, the lemma holds. □

Proof Sketch of Lemma 5.1: Firstly, we have $sc(u, v) \geq k - 1$ since u and v are in G which is $(k-1)$-edge connected. Secondly, $sc(u, v) < k$ since u and v are in different k-edge connected components. Thus, the lemma holds. □

Proof Sketch of Theorem 5.1: It is immediate to see that all $sc(u, v)$ values shall be assigned when Algorithm 6 terminates. Following from Lemma 5.1, we also know that all the assigned $sc(u, v)$ values are correct. Thus, the correctness of Algorithm 6 holds.

Now, we consider the time complexity. Let E_i denote the set of edges in $\phi_i(G)$ in Algorithm 6. Since the time complexity of ComputeKECCs (Algorithm 13) is $O(h \cdot l \cdot |E|)$, the time complexity of Algorithm 6 thus is $O(\sum_{k=1}^{|V|}(h \cdot l \cdot |E_i|)) = O(h \cdot l \cdot \sum_{k=1}^{|V|} |E_i|)$. For any edge $(u, v) \in E$ with $sc(u, v) = k$, it will appear in $E_1, \ldots, E_k$ but not in $E_i, \forall i > k$. Therefore, $\sum_{k=1}^{|V|} |E_i| = \sum_{(u,v) \in E} sc(u, v) \leq \sum_{(u,v) \in E} \lambda(u, v) \leq \sum_{(u,v) \in E} \min\{d(u), d(v)\} = \alpha(G) \cdot |E|$, where $d(\cdot)$ is the degree of a vertex and the last equation is proved in [10]. Thus, the time complexity of Algorithm 6 holds. □

Proof Sketch of Lemma 5.2: We can prove the lemma by contradiction. Assume that g^- is not $(k-1)$-edge connected, then there must exist a set of edges C in g^- whose removal disconnects g^- with $|C| \leq k - 2$. Also, we can see that the removal of $C \cup \{(u, v)\}$ disconnects g with $|C \cup \{(u, v)\}| \leq k - 1$; this contradicts that g is k-edge connected. Thus, the lemma holds. □

Proof Sketch of Lemma 5.3: Firstly, if $(u, v) \notin g$, then g^- is obviously a k-edge connected component of G^-. Secondly, if $(u, v) \in g$, then we have $g_{u,v} \subset g$ (i.e., $k \leq k_{u,v} - 1$) since $g \neq g_{u,v}$. From Lemma 5.2, we know that $g^-_{u,v}$ is $(k_{u,v} - 1)$-edge connected. Thus, g^- is k-edge connected since all edges except that in $g_{u,v}$ remain the same in g^-; moreover, g^- is a k-edge connected component of G^-. Thus, the lemma holds. □

Proof Sketch of Lemma 5.4: First of all, g^+ is k-edge connected. Thus, we only need to prove that there is no super graph g' of g^+ in G^+ that is also k-edge connected. We consider three cases.

1) $k < k_{u,v} + 1$. We prove it by contradiction. Assume that there is a super graph g' of g^+ in G^+ that is also k-edge connected. There are two cases. i) if $(u, v) \notin g'$, then g' is also k-edge connected in G and is a super graph of g. ii) if $(u, v) \in g'$, then $g^+_{u,v} \subseteq g'$; since $g_{u,v}$ is ($k_{u,v}$ ($\geq k$))-edge connected, g' by removing (u, v) is still k-edge connected and is a super graph of g. Both cases contradict that g is a k-edge connected component of G. Thus g^+ is a k-edge connected component of G^+.

2) $k > k_{u,v} + 1$. Obviously, $(u, v) \notin g^+$; otherwise g would be ($k-1$ ($> k_{u,v}$))-edge connected from Lemma 5.2, contradicting that $sc(u, v) = k_{u,v}$. Now, consider any super graph g' of g^+ in G^+. i) If $(u, v) \notin g'$, then g' is not k-edge connected, since $g \subset g' \subseteq G$ and g is a k-edge connected component of G. ii) If $(u, v) \in g'$, similarly we can prove that g' is not k-edge connected; otherwise, g' by removing (u, v) would be ($k-1$ ($> k_{u,v}$))-edge connected from Lemma 5.2 and is in G, contradicting that $sc(u, v) = k_{u,v}$. Thus, g^+ is a k-edge connected component of G^+.

3) $k = k_{u,v} + 1$ and $g^+ \not\subseteq g^+_{u,v}$ (or equivalently $g \not\subseteq g_{u,v}$). Now, consider any super graph g' of g^+ in G^+. i) if $(u, v) \notin g'$, then g' is not k-edge connected, since $g \subset g' \subseteq G$ and g is a k-edge connected component of G. ii) if $(u, v) \in g'$, then g' by removing (u, v) would be ($k-1$ ($= k_{u,v}$))-edge connected from Lemma 5.2; this contradicts that $g_{u,v}$ is a k-edge connected component of G with $k = k_{u,v}$ since $g \not\subset g_{u,v}$ and $g \subset g'$. Thus, $(u, v) \notin g'$ and g^+ is then a k-edge connected component of G^+.

Thus, the lemma holds. □

A.4 Additional Experiments

Description of Real Graphs. We included eleven real graphs in our testings, 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), Internet topology graph (as-Skitter), LiveJournal online social network (LiveJournal), the web graph within the .uk domain (uk-2002), and the twitter social network (twitter-2010). All datasets except uk-2002 and twitter-2010 are downloaded from the Stanford SNAP library[3], and detailed descriptions of these datasets can also be found there; the uk-2002 and twitter-2010 datasets are downloaded from LAW[4]. For each dataset, we extract the largest connected component in it as our test graph.

Graph	SC-MST∗	SC-MST	Graph	SC-MST∗	SC-MST
D5	0.01	2.05	D9	0.01	1.88
D6	0.01	1.68	D10	0.01	2.67
D7	0.01	0.93	D11	0.01	1.21
D8	0.01	0.87			
SSCA4	0.01	1.77	SSCA5	0.01	2.05

Table 10: Scalability testing for SC-MST∗ and SC-MST (in ms)

Graph	SMCC$_L$-OPT	Graph	SMCC$_L$-OPT
D5	16.8	D9	91
D6	8.66	D10	95
D7	1.39	D11	1.6
D8	22.4		
SSCA4	0.78	SSCA5	2.49

Table 11: Scalability testing for SMCC$_L$-OPT (in seconds)

[3] http://snap.stanford.edu/data/
[4] http://law.di.unimi.it/datasets.php

A.5 Compute k-Edge Connected Components

For completeness, we also briefly present the state-of-the-art approach for computing k-edge connected components in [7].

Algorithm 13: ComputeKECCs[7]

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

```
1  G_g ← Decompose(G, k);                    /* Decompose G */;
2  if G_g consists of only one subgraph then φ_k(G) ← φ_k(G) ∪ G_g;
3  else for each subgraph g in G_g do ComputeKECCs(g, k);
4  Procedure Decompose(G, k)
5      G_g ← ∅; PG ← G;
6      while PG is non-empty do
7          Compute a maximum adjacency order L of vertices in PG;
8          for each vertex u in L with w(L, u) ≥ k do
9              Merge u with the immediately preceding vertex of u in L into
               a single super-vertex;
10         while the last vertex v in L has w(L, v) < k do
11             Remove v from both L and PG and add to G_g the subgraph of
               G induced by vertices contained in the super-vertex v;
12     return G_g;
```

Framework. Based on the property that a graph G is k-edge connected if and only if every cut has at least k edges, a decomposition-based framework was proposed in [7] as shown in Algorithm 13 which is self-explanatory. The main component is Decompose, which decomposes an input graph G into a set of disjoint, vertex-induced, connected subgraphs of G such that the union of their vertices is the set of vertices of G. To ensure the correctness, Decompose must satisfy two properties [7]. *1) Atomicity property*: given an input graph G, each k-edge connected component of G is entirely contained in one of the returned subgraphs by Decompose. *2) Cutability property*: if the input graph G is not k-edge connected, then Decompose decomposes G into at least two subgraphs.

Decompose. The procedure Decompose decomposes a graph G by recursively removing from G the edges that are in cuts of G with cardinality less than k, where a *cut* is a subset of E whose removal disconnects G. Such cuts are found by the *maximum adjacency search* [7, 28], which computes a maximum adjacency order L of all vertices in G as follows. L is initialized to contain a single vertex arbitrarily chosen from V. As long as there are vertices not included in L, it adds to the tail of L the vertex u 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 vertices in L.

Lemma A.3: *[7] Given a graph G with a computed maximum adjacency order L, (case-i) for any vertex u in L, if $w(L, u) \geq k$, then the edge connectivity between p_u and u in G is at least k (i.e., $\lambda(p_u, u) \geq k$), where p_u is the immediately preceding vertex of u in L; (case-ii) for the last vertex v in L, if $w(L, v) < k$, then $\lambda(u, v) < k$ for any vertex $u \neq v$ in L.* □

The pseudocode of Decompose is shown in Algorithm 13. PG is a partition graph [7] initialized as G, where each vertex in PG is a super-vertex containing a set of vertices in G and the sets of vertices in the super-vertices of PG form a partition of the vertices of G. Line 7 computes L, and Lines 10–11 and Lines 12–14 correspond to case-i and case-ii in Lemma A.3, respectively.

Time Complexity. Let h denote the maximum depth of recursively calling ComputeKECCs in Algorithm 13, and l denote the number of iterations in Decompose (i.e., Line 8) [7]. Then, the time complexity of Algorithm 13 is $O(h \cdot l \cdot |E|)$ since Line 9 takes $O(|E|)$ time, where h and l are shown to be bounded by small constants for real graphs [7].