On Minimal Steiner Maximum-Connected Subgraph Queries

Jiafeng Hu, Xiaowei Wu, Reynold Cheng, *Member, IEEE*, Siqiang Luo, and Yixiang Fang

Abstract—Given a graph G and a set Q of query nodes, we examine the *Steiner Maximum-Connected Subgraph* (SMCS) problem. The SMCS, or G's induced subgraph that contains Q with the largest connectivity, can be useful for customer prediction, product promotion, and team assembling. Despite its importance, the SMCS problem has only been recently studied. Existing solutions evaluate the *maximum SMCS*, whose number of nodes is the largest among all the SMCSs of Q. However, the maximum SMCS, which may contain a lot of nodes, can be difficult to interpret. In this paper, we investigate the *minimal SMCS*, which is the minimal subgraph of G with the maximum connectivity containing Q. The minimal SMCS contains much fewer nodes than its maximum counterpart, and is thus easier to be understood. However, the minimal SMCS can be costly to evaluate. We thus propose efficient Expand-Refine algorithms, as well as their approximate versions with accuracy guarantees. We further develop a cache-based processing model to improve the efficiency for an important case when Q consists of a single node. Extensive experiments on large real and synthetic graph datasets validate the effectiveness and efficiency of our approaches.

Index Terms—Community search, minimal steiner maximum-connected subgraph, k-edge connectivity, subgraph search

1 Introduction

Graphs are prevalent in various domains, such as social science, e-commerce, and biology. Given a graph G and a set Q of nodes, we study the *Steiner Maximum-Connected Subgraph* (or *SMCS*), a subgraph of G with the maximum connectivity that contains Q. The SMCS can be used in customer prediction, community search, product promotion, and team assembling [6]. In a social network (e.g., Facebook), given a set Q of nodes denoting social network users, its SMCS represents a group of people with similar interest. The members of the SMCS found can then be considered for product recommendation. As another example, in a Protein-Protein-Interaction (PPI) network [3], the SMCS can be used to discover a subgraph connecting a given set Q of protein nodes; the protein nodes appearing in the SMCS can have a close relationship. In a bibliographic network (e.g., DBLP), the SMCS can be used to look for research communities, in order to facilitate collaboration. Fig. 1 illustrates an SMCS for the set Q = {"Michael Stonebraker", "Samuel Madden", "Daniel J. Abadi", "Jennie Duggan"}, extracted from the DBLP. This SMCS illustrates the researchers who are related to those specified in Q.

The notion of the SMCS has only been recently studied. In particular, Chang et al. [6] investigated a variant of the SMCS (or known as the *Steiner Maximum-Connected Component* (SMCC) in [6]), which is the *maximum SMCS* whose number of nodes is the largest among all the possible SMCSs. We have tested the solution provided by [6] on some datasets. We found that the maximum SMCS has a high cohesiveness (because its *connectivity*, or the smallest number of edges whose removal disconnects it, is maximized). Unfortunately, the maximum SMCS is often extremely large and complex. On the DBLP dataset that contains 803 K nodes and 3.2 M edges, the average number of nodes of a maximum SMCS is over 400 K. This not only hinders the analysis of the SMCS structure, but also makes it difficult to be used in real situations. Suppose that a user wants to set up an academic conference. She has a small budget to invite a few renowned scholars and their related researchers. To decide the invitation list, the user may issue a maximum SMCS query, with Q containing the names of several researchers, on DBLP. She can then contact the researchers (or graph nodes) that appear in the SMCS. However, if the maximum SMCS is very large, the user can have a hard time to figure out the appropriate participants. Is it possible to get a *smaller* SMCS, while maintaining the maximum connectivity?

In this paper, we examine the discovery of an SMCS that has a small number of nodes. One way is to evaluate the *minimum SMCS*, whose number of nodes is the smallest among all the possible SMCSs. However, as we will discuss in Section 3.2, finding the minimum SMCS is NP-hard. Furthermore, it is NP-hard to get an approximate minimum SMCS with any constant ratio. Thus, any attempt to obtain a minimum SMCS or its approximate version appears to be a futile exercise. We study another version of SMCS, called the *minimal SMCS*, which is essentially an SMCS of Q (denoted as G'), such that any subgraph of G' containing Q is *not* an SMCS of Q. While the minimal SMCS is still challenging to find, we show that it can be derived in polynomial time. To our understanding, the evaluation of the minimum and minimal SMCS's have not been studied before.

• *The authors are with the Department of Computer Science, University of Hong Kong, Pokfulam Road, Hong Kong.*
E-mail: {jhu, xwwu, ckcheng, sqluo, yxfang}@cs.hku.hk.

Manuscript received 18 Oct. 2016; revised 11 July 2017; accepted 13 July 2017. Date of publication 24 July 2017; date of current version 4 Oct. 2017.
(Corresponding author: Jiafeng Hu.)
Recommended for acceptance by K.S. Candan.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TKDE.2017.2730873

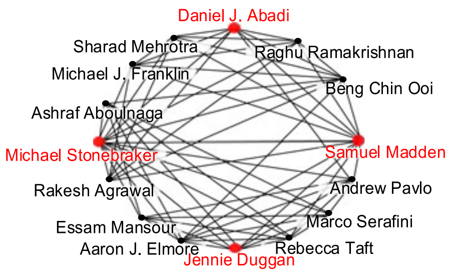


Fig. 1. A minimal SMCS of the DBLP using query Q = {"Michael Stonebraker", "Samuel Madden", "Daniel J. Abadi", "Jennie Duggan"}.

Example 1.1. Fig. 2 illustrates three SMCSs, namely, G_1, G_2, and G_3, for a graph G and query node set $Q = \{f\}$. All these SMCSs have a maximum connectivity of 3, i.e., at least 3 edges have to be removed in order to disconnect them. Here, G_3 is the maximum SMCS, since it has the largest number of nodes, while G_2 is the minimum one. Both G_1 and G_2 are minimal SMCSs.

Because obtaining the minimum SMCS is computationally intractable, we study the efficient retrieval of the minimal SMCS. One simple way is to first adopt the solution in [6] to compute the maximum SMCS G', and then iteratively refine G' to ensure its minimality. While this solution is simple, it has a high complexity, since the cost of testing the minimality of an SMCS is high (Section 4.2.1). Moreover, due to the huge size of the maximum SMCS, it is extremely slow in our experiments (Section 7).

Our Contributions. We have designed a minimal SMCS solution called the *Expand-Refine* framework. In the *Expand* step, through local expansion of nodes starting from nodes in Q, we obtain a subgraph G' of G, which satisfies the requirement of an SMCS. Intuitively, we obtain G' by exploring the neighboring nodes of Q until we obtain an SMCS. In the *Refine* step, we devise an efficient algorithm that removes nodes based on the dependence of nodes on their minimal SMCSs.

We further improve the efficiency of solutions by relaxing the constraints from two perspectives, namely connectivity and minimality. In the *Expand* step, we bound the expansion space (to relax the connectivity); in the *Refine* step, we develop an approximation solution with accuracy guarantees (to relax the minimality).

Moreover, we observe that in many applications, it is interesting to find the minimal SMCS of a single node (i.e., $|Q| = 1$) [16], [22]. For example, in friend recommendation, a social network user u may request for new friends; the application can issue a minimal SMCS query with $Q = \{u\}$. In e-commerce network systems (e.g., eBay), new products can be recommended to a user, where a minimal SMCS query is issued based on the graph node representing the user. We thus develop a minimal SMCS algorithm customized to single-node queries. The solution saves the processing information related to the current query in a small cache structure, and use them to answer the subsequent ones. Our results show that the single-node solution performs well for random single-node queries. We further develop an approximate version for this solution.

We have performed a detailed evaluation of our algorithms on large real and synthetic graph datasets. Our experimental results confirm our claim that the minimal SMCS has

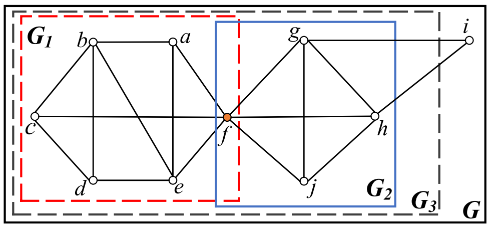


Fig. 2. The maximum SMCS (G_3), minimum SMCS (G_2), and minimal SMCSs (G_1 and G_2) for query Q = {f}.

a higher "quality" than the maximum SMCS, the k-core model (GrCon [4]) and the closest k-truss model (LCTC [24]), in terms of fewer nodes, and higher edge density and connectivity. The efficiency of our new minimal SMCS solutions addresses several orders of magnitude improvement over basic solutions. Our customized solution for single-node SMCS queries demonstrates further improvement in speed.

Organization. We review the related work in Section 2. Section 3 formulates the SMCS and analyses the minimum SMCS problem. Section 4 discusses our Expand-Refine solutions. In Section 5, we present our relaxation strategies to further improve the efficiency. In Section 6, we present our solutions for single-node minimal SMCS queries. We report our experimental results in Section 7. Section 8 concludes.

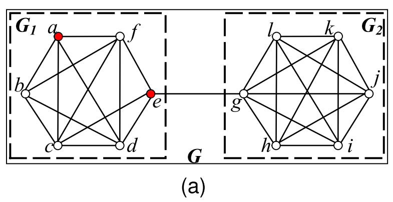
2 Related Work

Our work is related to topics of *connectivity*, *community search*, and *cohesive subgraph detection*.

Connectivity. The SMCS is a subgraph of G with the maximum *edge-connectivity* (called *connectivity* here). The connectivity of a graph G is the minimum number of edges whose removal disconnects G [20]. Connectivity has been studied in a wide range of graph-related problems, including network reliability [14], VLSI chip design [29], transportation planning [5], social networks [37], computational biology [34], and cohesive subgraph detection [1], [8]. However, it has only been recently used to facilitate the search of cohesive subgraphs for a given set of nodes (or SMCS) [6]. In this paper, we study the SMCS problem extensively.

Community Search. Given a set Q of nodes in a graph G, the community search problem aims at finding the subgraphs of G that contains Q. For this problem, various goodness metrics have been proposed, including local modularity [12], minimum degree [4], [16], [33], trussness [22], [24] (the minimum support of an edge in the subgraph, where the support of an edge is the number of triangles containing it), α-adjacency-γ-quasi-k-clique [15], quasi-clique [26], query biased edge density [38], attributed community [18], [19], [23] and spatial-aware community [17]. These measures are fundamentally different from connectivity, and so their solutions cannot be used to obtain the SMCS.

Moreover, as mentioned in [1], [27], compared with minimum degree and trussness, connectivity is a better cohesiveness metric. In particular, the minimum degree only restricts degrees of nodes in subgraphs without any structure constraint [1]. In Fig. 3a, with query node set $Q = \{a, e\}$, the whole graph G will be returned under the minimum degree metric (where the minimum degree is maximized). However, under the connectivity metric, a better subgraph G_1 in which all nodes are highly connected will be returned, since the nodes in G_2 are far away from query nodes. As for the trussness measure, it can be too restrictive on the triangle structure,



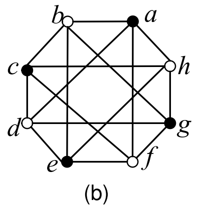


Fig. 3. (a) An ill-connected graph that minimum degree fails to separate for query Q = {a, e}. (b) A well-connected bipartite graph that cannot be found with trussness [1] for query Q = {c}.

which is a local concept, whereas connectivity is more global [1]. Notice that there is no triangle in bipartite graphs, e.g., paper-author graphs, online dating graphs, or product-purchaser graphs. For these graphs, it is better to use connectivity as a goodness metric, since no cohesive subgraphs with trussness can be found. For example, in Fig. 3b, with query node set $Q = \{c\}$, no subgraph will be returned under trussness metric, because there is no triangle in the subgraph. On the other hand, the whole graph (with connectivity equal to 4) is returned under the connectivity metric. Hence, in this paper, we use connectivity for community search, and develop solutions for obtaining the minimal SMCS.

Compared with our previous work [21], we make the following significant additions. First, we developed a minimal SMCS algorithm customized to single-node queries, as well as an approximation algorithm. Second, we conducted additional experiments on large real graph datasets to evaluate the customized solution for single-node queries and its approximate version. Third, we performed additional experiments to compare our solutions with two state-of-the-art community search models, i.e., the closest k-truss community model (LCTC) [24] and the cohesive k-core community model (GrCon) [4].

Cohesive Subgraph Detection. In recent years, there has been a lot of work on the retrieval of cohesive subgraphs from a large graph. Different kinds of cohesive subgraphs have been studied, such as the maximal clique [7], [10], quasi-clique [39], k-plex [36], k-core [9], [28], [32], k-truss [13], [35], and locally densest subgraph [31]. However, these solutions are inherently different from our problem, in which they are not query-dependent (i.e., the specification of Q is not required). Hence, their techniques are inapplicable to compute minimal SMCS.

3 The SMCS Problem

We will describe the graph model and three types of SMCSs in Section 3.1. We study the intractability problem of the minimum SMCS in Section 3.2.

3.1 Connectivity and SMCS

Given an undirected graph G, let $V(G)$ and $E(G)$ be its sets of nodes and edges respectively. We use $G[S] = (S, E[S])$ to denote the subgraph of G induced by node set $S \subseteq V(G)$, where $E[S] = \{(u,v) \in E(G) : u, v \in S\}$. Let $N(u)$ be the set of neighbors of u in G. We denote by $G\backslash u$ the graph obtained by removing node u from $V(G)$. We use *component* to refer to a connected component of G.

Definition 3.1 (Connectivity). *The connectivity (or edge-connectivity in [20]) $\lambda(u,v)$ between two distinct nodes u and v in $V(G)$ is the minimum number of edges whose removal disconnects u and v. The connectivity of the graph $\lambda(G) = \min_{u,v \in V(G)} \lambda(u,v)$ is the minimum connectivity between any two distinct nodes in G (i.e., the smallest number of edges whose removal disconnects G).*

Definition 3.2 (k-Component). *A subgraph g of G is a k-component (or k-edge connected component in [1], [8]) if 1) $\lambda(g) \geq k$; and 2) the connectivity of any super-graph of g in G is less than k.*

Given a set of query nodes $Q \subseteq V$, we use G_Q to denote a node-induced subgraph of G containing Q. We define the SMCS as follows.

Definition 3.3 (SMCS). *The Steiner Maximum-Connected Subgraph is a subgraph G_Q of G such that the connectivity $\lambda(G_Q)$ of G_Q is maximized.*

Let $sc(Q)$ be the connectivity of any SMCS of Q. In [6], $sc(Q)$ is called the *Steiner-connectivity* of Q. When $Q = \{u\}$, we use $sc(u)$ to denote $sc(\{u\})$.

Definition 3.4 (Maximum SMCS). *An SMCS G_Q of Q such that the number of nodes in G_Q is maximized.*

The maximum SMCS is also called the *Steiner Maximum-Connected Component* in [6]. In our experiments, the maximum SMCS suffers from having a huge size and low edge density. Hence, we study the problems of finding other SMCS alternatives as follows.

Definition 3.5 (Minimum SMCS). *An SMCS G_Q of Q such that the number of nodes in G_Q is minimized.*

Definition 3.6 (Minimal SMCS). *An SMCS G_Q of Q such that any proper induced subgraph of G_Q containing Q is not an SMCS of Q.*

Note that a minimum SMCS is also a minimal SMCS. Fig. 2 shows these three kinds of SMCSs. Next, we discuss the problem of finding the minimum SMCS.

3.2 Intractability of Minimum SMCS

It is easy to observe that the minimum SMCS problem is APX-hard since it is a generalization of the `Steiner tree` problem: given any subset of nodes S in $G(V,E)$, a minimum subgraph spanning all nodes in S can be found by computing a minimum SMCS of $S \cup \{u\}$ in $G(V \cup \{u\}, E \cup \{(s,u)\})$, where $s \in S$ and $u \notin V$. Note that although the objectives are slightly different (one aims at minimizing the number of edges while the other aims at minimizing the number of nodes), we can create dummy nodes within each edge to make the two objectives arbitrarily close. Since the `Steiner tree` problem is APX-hard, the minimum SMCS is also APX-hard. We further show in the following that the problem does not admit any constant approximation ratio, even when restricted to the case when $|Q| = 1$. The reduction from `vertex cover` problem is a modification of the hardness proof of the MSMD_3 problem in [2]. The detailed proof of Theorem 3.1 can be found in our previous work [21].

Theorem 3.1 (Inapproximability). *Unless* P = NP, *there does not exist any polynomial-time algorithm that approximates the minimum SMCS problem within any constant ratio.*

Theorem 3.1 states that it is not only intractable to obtain a minimum SMCS, but also hard to get its approximate version in an accurate manner. Hence, we focus on the minimal SMCS, which although may be larger than the minimum SMCS, can be found in polynomial time.

4 Processing Minimal SMCS

We now examine how to find a minimal SMCS from G efficiently. Let us first present our *Expand-Refine* framework. As shown in Algorithm 1, it contains three steps. First, we compute the Steiner-connectivity $sc(Q)$ of the query node set Q. We then perform Expand (line 2) to generate an SMCS of Q, which serves as a candidate of the minimal SMCS. We then execute Refine (line 3) on the SMCS returned by line 2. This operation removes selected nodes and returns a minimal SMCS of Q.

Algorithm 1. Expand-Refine(G, Q)

```
1 Compute the Steiner-connectivity $sc(Q)$ of $Q$;
2 $H \leftarrow$ Expand($(V_{sc(Q)}, E_{sc(Q)}), Q, sc(Q)$);
3 $H_Q \leftarrow$ Refine($G[H], Q, sc(Q)$);
4 return $H_Q$;
```

Step 1: Find $sc(Q)$. An efficient algorithm for obtaining $sc(Q)$ has been proposed in [6]. The solution is based on the *connectivity graph*. Let us use $sc(u, v)$ to denote $sc(\{u, v\})$. If $(u, v) \in E$, then $sc(u, v)$ is the maximum connectivity of any graph containing edge (u, v).

Definition 4.1 (Connectivity Graph [6]). *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 nodes and edges as G. Each edge (u, v) in G_c has a weight $w(u, v)$ equal to the Steiner-connectivity $sc(u, v)$ of $\{u, v\}$ in G.*

An efficient algorithm was proposed in [6] to compute G_c in $O(\alpha(G) \cdot h \cdot l \cdot |E|)$ time, where $\alpha(G)$ is the "arboricity" of G and is bounded by (usually much smaller than) $\sqrt{|E|}$ [11], and h and l are usually bounded by small constants for real graphs [8]. Based on G_c, [6] developed an $O(|Q|)$ algorithm to compute the Steiner-connectivity $sc(Q)$. For a single node u, its Steiner-connectivity is $sc(u) = \max_{v \in N(u)} sc(u, v)$, i.e., the maximum weight among all edges adjacent to u in G_c.

From now on, we assume that G_c has been computed. Thus, the $sc(u, v)$ and $sc(u)$ values for every edge $(u, v) \in E$ and node $u \in V$ are readily known. In the rest of this section, we will explain the details of Expand (Section 4.1) and Refine (Section 4.2).

4.1 The Expand Operation (Algorithm 1, Step 2)

The goal of this step is to return a candidate for minimal SMCS of Q. In fact, any SMCS of Q can be a candidate, and one simple way is to obtain the maximum SMCS, using the solution provided by [6]. However, as we will show in our experiments, the maximum SMCS computed can be extremely large. If the maximum SMCS of Q is returned in this step, the efficiency of Refine (Step 3 of Algorithm 1) can be seriously affected - a huge number of nodes have to be removed before we can get the minimal SMCS of Q. We next present a better method that generates a smaller SMCS of Q efficiently. To start with, let us present the notion of *layer*.

Definition 4.2 (Layer). *For all $k \geq 1$, let $V_k = \{u \in V : sc(u) \geq k\}$ and $E_k = \{(u, v) \in E : sc(u, v) \geq k\}$. We call $G_k = (V_k, E_k)$ the kth layer of graph G.*

Given an integer k, G_k can be obtained easily. We first compute E_k, which contains all the edges in G_c whose sc values are higher than k. Then, the set V_k is simply the set of nodes induced by E_k. We have developed a useful lemma for G_k.

Lemma 4.1. *For all $k \geq 1$, each component of the kth layer $G_k = (V_k, E_k)$ is k-connected.*

Proof. By definition, we know that for any edge (u, v) in the component G' of (V_k, E_k), there must exists a subgraph G_{uv} of G' containing (u, v) that is k-connected. Hence any two adjacent nodes in G' are k-connected, implying that G' is k-connected. □

We now describe our *local expansion* strategy, which finds a subgraph of G containing Q that is $sc(Q)$-connected. Let $k = sc(Q)$ be obtained in Step 1 of Algorithm 1. Given the kth layer G_k=(V_k, E_k) and a set Q of query nodes, Algorithm 2 computes an SMCS of Q by performing a local search on G_k. Particularly, we first form a Steiner tree on the graph G_k to connect all query nodes (line 2). Since the Steiner tree problem is NP-hard, a well-known 2-approximation algorithm [30] is adopted to construct the Steiner tree. We then iteratively expand the candidate node set S by involving the local neighbors of the query nodes in a breadth-first-search manner and invoke ComputeKECC to test whether there exists a k-component containing Q in $G[S]$, until a valid SMCS of Q is found (lines 3-6). Here, function ComputeKECC returns the node set of that component if it exists, or an empty set otherwise (line 6). To implement this function, we first invoke the best k-component algorithm, called KECCs-Exact [8], which returns all the k-components of $G_k[S]$ in $O(h \cdot l \cdot |E|)$ time, where h and l are usually bounded by small constants.

Algorithm 2. Expand(G_k, Q, k)

```
Input: A graph $G_k = (V_k, E_k)$, a set $Q$ of query nodes, and the
       Steiner-connectivity $k$ of $Q$
Output: The node set of an SMCS of $Q$
1 $S \leftarrow \emptyset$, $H \leftarrow \emptyset$;
2 Compute a Steiner Tree $S$ from $G_k$ containing $Q$;
3 while $H = \emptyset$ do
4     pick $u \in S$ closest to $Q$ such that $N(u) \nsubseteq S$;
5     $S \leftarrow S \cup N(u)$;
6     $H \leftarrow$ ComputeKECC($G_k[S], Q, k$);
7 return $H$;
```

Then ComputeKECC returns the k-component that contains Q. For convenience, we use $\mathsf{M} = O(h \cdot l \cdot |E|)$ to denote the time complexity of ComputeKECC.

Algorithm 2 always returns an SMCS of Q, since in the worst case, the maximum SMCS will be returned. In our experiments, the number of nodes in SMCSs returned is much smaller than that of their maximum counterparts.

4.2 The Refine Operation (Algorithm 1, Step 3)

After obtaining an SMCS $G[H]$ of Q with $sc(Q)$ = k in Step 2, we need to check its minimality, i.e., any subgraph induced by a proper subset of H is not an SMCS of Q. However, since there are $2^{|H|-|Q|}$ possible induced subgraphs of H containing Q, examining all possible subgraphs is not feasible. To validate the minimality of the SMCS, we first describe a basic refinement algorithm in Section 4.2.1, and then propose a better solution in Section 4.2.2. To further improve the efficiency, we introduce the incremental removal optimization technique in Section 4.2.3.

4.2.1 Basic Refinement

If $G[H]$ is not minimal, then there must exist a subgraph $G[H_Q]$, where $H_Q \subsetneq H$, $Q \subseteq H_Q$ and $\lambda(G[H_Q]) = k$, i.e., there exists a k-component containing Q if any node in $H\backslash H_Q$ is removed. Based on this intuition, we develop the Refine-Basic Algorithm. Each node in $H\backslash Q$ is tested iteratively. If a smaller SMCS is found, we shrink the candidate set and recursively call Refine-Basic to find a minimal SMCS. The node set H of a minimal SMCS will be returned if there does not exist a k-component containing Q when any node in $H\backslash Q$ is removed.

Theorem 4.1. *Refine-Basic returns a minimal SMCS of Q in $O(|H|^2\mathsf{M})$ time.*

Please refer to our previous work [21] for the proof.

4.2.2 Advanced Refinement

We now explain how to use the relationship between nodes to speed up the refinement step.

Definition 4.3 (Separable). *Given any SMCS $G[H]$ of a set Q of nodes, node $u \in H\backslash Q$ is* separable *for Q if there exists an $sc(Q)$-component containing Q in $G[H]\backslash u$.*

Lemma 4.2. *Given any SMCS $G[H]$ of a set Q of nodes, if $u \in H\backslash Q$ is non-separable for Q, then any SMCS of Q that is a subgraph of $G[H]$ must contain u.*

Please refer to our previous work [21] for the proof.

Given an SMCS $G[H]$ of Q, Algorithm 3 computes a minimal SMCS of Q by repeatedly removing separable nodes from the SMCS. We initialize T as the set of nodes whose separabilities are not yet tested (line 1). Then in every round we test the separability of a node $u \in T$ (lines 2-8). If u is non-separable, it will be removed from T (line 6); otherwise it is separable, then we can shrink H to a proper subset $H' \subsetneq H$ that does not contain u and update T (line 8). Note that we can remove at least one node from T in each step and when T is empty, all nodes except those in Q in the current SMCS must be non-separable, which ensures the minimality.

Theorem 4.2 (Minimal-SMCS). *Given an SMCS $G[H]$ of a set Q of nodes, Refine$(G[H], Q, k)$ computes a minimal SMCS of Q that is a subgraph of $G[H]$ in $O(t \cdot \mathsf{M})$ time, where t is the number of iterations ($t < |H|$).*

Please refer to our previous work [21] for the proof. Given the above algorithms, a minimal SMCS of Q can be computed by calling Algorithm 1 with inputs G and Q.

Algorithm 3. Refine$(G[H], Q, k)$

```
1 $T \leftarrow H\backslash Q$;
2 while $T \neq \emptyset$ do
3     pick a node $u \in T$;
4     $H' \leftarrow$ ComputeKECC$(G[H]\backslash u, Q, k)$;
5     if $H' = \emptyset$ then                /* $u$ is non-separable */
6         $T \leftarrow T\backslash u$;
7     else                                   /* $G[H']$ is an SMCS of $Q$ */
8         $H \leftarrow H', T \leftarrow T \cap H'$;
9 return $H$;
```

4.2.3 Incremental Removal Optimization

Note that given an SMCS $G[H]$ of a set Q of query nodes with $sc(Q) = k$, in the refinement step, we identify the separability of a sample node in each iteration. If the sampled node u is separable, then all nodes not contained in the k-component containing Q in $G[H]\backslash u$ are separable. However, in some cases, especially when k is small, every time when a separable node is identified, we can only reduce the size of SMCS by a small constant, which is inefficient when the candidate SMCS is large.

One observation is, for any subset $U \subseteq H\backslash Q$, if there exists a k-component containing Q after removing U from $G[H]$, then we can identify all nodes in U separable immediately. Using this idea, we develop an incremental removal optimization for the refinement algorithm, as shown in Algorithm 4. The key idea is that we try to sample a set of nodes in each iteration. Let i denote the number of nodes to be sampled, initialized to 1 (line 2). After one successful removal during the iteration procedure (a k-component containing Q is found after removing U), the sample size i will be increased by 1 (line 12). When it meets an unsuccessful removal (there does not exist a k-component containing Q after removing U), if the size of U is 1, the only element in U will be removed from T since it is non-separable (line 8); otherwise, the sample size i will be reset to 1 (line 10). Note that in the second case, the nodes in U cannot be labeled as non-separable, even though we know one of them is.

Algorithm 4. Refine-Inc$(G[H], Q, k)$

```
1 $T \leftarrow H\backslash Q$;
2 $i \leftarrow 1$;
3 while $T \neq \emptyset$ do
4     $i \leftarrow \min(i, |T|)$;
5     sample $i$ nodes $U$ from $T$;
6     $H' \leftarrow$ ComputeKECC$(G[H\backslash U], Q, k)$;
7     if $H' = \emptyset$ and $i = 1$ then
8         $T \leftarrow T\backslash U$;
9     else if $H' = \emptyset$ and $i > 1$ then
10        $i \leftarrow 1$;
11    else
12        $H \leftarrow H', T \leftarrow T \cap H', i \leftarrow i + 1$;
13 return $H$;
```

5 Improving Performance by Constraint Relaxation

In this section, we focus on improving the efficiency of our Expand-Refine algorithm in two ways. First, for the *Expand* operation, we constrain the local search space while relaxing the connectivity. Second, for the *Refine* operation, we propose an approximation algorithm to speed up the refinement procedure with accuracy guarantee.

5.1 Early Stop in the Expand Step

As described in Section 4.1, in the *Expand* step, we locally expand the Steiner tree S to find an SMCS of Q. However, since the local expansion is a heuristic search strategy, the time cost may be very high in some cases, especially when the graph size is very large. In order to reduce the search space, we use a threshold θ to bound the size of S, so called *early stop*. Specifically, in the Bounded-Expand Algorithm, after obtaining the Steiner tree S for the query nodes, we expand the tree S to a graph in a BFS manner until the node size exceeds a threshold θ, i.e., $|S| > \theta$, where θ is empirically tuned. Then we extract a k'-component H containing Q in $G_k[S]$, where $k' \leq k$ is the maximum possible connectivity. Following that, we may get a candidate subgraph with connectivity less than the Steiner-connectivity of Q. However, as we will show in Section 7.3, the connectivity k' of the subgraph returned by Bounded-Expand is very close to the corresponding maximum one in practice.

5.2 Approximation in the Refine Step

Note that to ensure an SMCS graph $G[H]$ of a set Q of nodes is minimal, we need to test separability for each node $u \in H\backslash Q$, which takes $\Omega(|H| \cdot \mathsf{M})$ time in the worst case. For a large k, usually we have a large minimal SMCS, which leads to long processing time. To further improve the efficiency of the refinement procedure, we propose an approximation algorithm which incorporates two extra user-specified parameters into the input, i.e., the approximation ratio r and the failure probability δ. The approximation algorithm stops earlier and outputs with probability at least $(1-\delta)$ an SMCS of Q that is an r-approximation of a minimal SMCS of Q, for any constant $\delta \in (0,1)$ and $r > 1$. Note that an SMCS $G[H]$ of Q is an r-approximation if there exists $H_Q \subseteq H$ such that $|H_Q| \geq \frac{1}{r}|H|$ and $G[H_Q]$ is a minimal SMCS of Q. The failure probability δ is defined as the probability that the approximation ratio is larger than r.

The pseudocode of the approximation method is shown in Algorithm 5. Different from Algorithm 3, we maintain an extra variable, i.e., the number of non-separable nodes sampled consecutively, denoted as *step*. At first, the value of *step* is initialized to 0 (line 2). In each iteration of the algorithm, we sample a node u from T uniformly at random (line 4) and test its separability (lines 5-9). If u is non-separable (line 6), then it will be removed from T and the value of *step* will be increased by 1 (line 7); otherwise we can shrink H to the k-component containing Q in $G[H]\backslash u$, update T and reset the value of *step* to 0 (line 9). The iteration will be halted when the value of *step* is not less than $\omega = \lceil \frac{\log \frac{1}{\delta}}{\log r} \rceil$ or T is empty. Lemma 5.1 shows that our approximation algorithm outputs an r-approximation of a minimal SMCS of Q in $G[H]$ with probability at least $(1-\delta)$. Please refer to our previous work [21] for the proof and detailed examples.

Algorithm 5. Approx-Refine($G[H]$, Q, k, r, δ)

```
1  $T \leftarrow H\backslash Q, \omega = \lceil \frac{\log \frac{1}{\delta}}{\log r} \rceil$;
2  $step \leftarrow 0$;
3  while $step < \omega$ and $T \neq \emptyset$ do
4      sample uniformly at random a node $u \in T$;
5      $H' \leftarrow$ ComputeKECC($G[H]\backslash u, Q, k$);
6      if $H' = \emptyset$ then
7          $T \leftarrow T\backslash u, step \leftarrow step + 1$;
8      else
9          $H \leftarrow H', T \leftarrow T \cap H', step \leftarrow 0$;
10 return $H$;
```

Lemma 5.1 (Approximate Minimal SMCS). *Given an SMCS $G[H]$ of a set Q of nodes, for any constant $\delta \in (0,1)$ and $r > 1$, Algorithm 5 returns an r-approximation of a minimal SMCS of Q in $G[H]$ with probability at least $(1-\delta)$.*

The incremental removal optimization can also be used in the approximation algorithm. We call it Approx-Refine-Inc algorithm. The main idea is similar to Algorithm 4. Whenever we sample a separable node, the sample size i is increased by 1, otherwise it will be set to 1. Since our algorithm stops only when ω consecutive non-separable nodes are sampled (which means the sample size is always 1), Lemma 5.1 still holds for the Approx-Refine-Inc algorithm.

6 Optimizing Single-Node Queries

In many applications, like friend recommendation on social networks (e.g., Facebook), product promotion on e-commerce networks (e.g., eBay), etc., single-node minimal SMCS queries ($|Q| = 1$) account for a large proportion among all queries. In the meantime, a large number of queries will be frequently triggered in those applications. In order to speed up single-node minimal SMCS queries, we discuss in this section how to utilize the minimal SMCSs returned for future queries. In Section 6.1, we propose a cache-based processing model for single-node queries. Then, in Section 6.2, we adopt the approximation technique introduced in Section 5.2 to the cache-based processing model to further improve the efficiency.

6.1 Cache-Based Processing Model

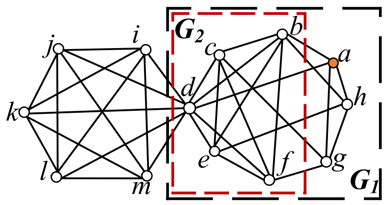
Although the minimal SMCS for each query node can be totally different, if we have already computed the minimal SMCS $G[H_u]$ for node u, then for all nodes in H_u with the same Steiner-connectivity as u, $G[H_u]$ which is expected to be small, can be taken as a candidate SMCS. Based on this property, we propose a cache-based processing model for single-node queries to further improve the efficiency.

Algorithm 6. Cache-Minimal-SMCS(G)

```
1  compute the connectivity graph of $G$;
2  for all $u \in V$, min-smcs($u$) $\leftarrow$ NULL;
3  for each single-node query $Q = \{u\}$ do
4      if min-smcs($u$) $\neq$ NULL then
5          output min-smcs($u$);
6      else
7          $H \leftarrow$ Expand($(V_k, E_k), u, sc(u)$);
8          output $H_u \leftarrow$ Refine($G[H], u, sc(u)$)
9          min-smcs($u$) $\leftarrow G[H_u]$;
10         if $|H_u| \leq \eta$ then
11             $S(H_u) \leftarrow \{v \in H_u$ :
               min-smcs($v$) = NULL, $sc(v) = sc(u)\}$;
12             Find-Minimal($G[H_u], S(H_u), H_u, sc(u)$);
               */ details in Section 6.1.2 */
```

Our cache-based processing model is outlined in Algorithm 6. The key idea is as follows. The algorithm maintains a pointer min-smcs for each node $u \in V$, which points to a minimal SMCS of u if its minimal SMCS has been calculated, or NULL otherwise (all pointers are initialized to NULL at the beginning (line 2)). When a single-node query $Q = \{u\}$ arrives, we first check whether the pointer min-smcs(u) is NULL or not. If min-smcs(u) $\neq$ NULL, then the minimal SMCS pointed to by min-smcs(u) will be returned directly (line 5). Otherwise, following steps will be executed:

1) calculate the minimal SMCS $G[H_u]$ of u using Algorithm 1 introduced in Section 4 (lines 7-8);
2) update min-smcs(u) to $G[H_u]$ (line 9);
3) if the size of H_u is no larger than a threshold η, then:



(a) A graph with Q={a}

Id	sc	Id	sc
a	4	h	4
b	4	i	5
c	4	j	5
d	5	k	5
e	4	l	5
f	4	m	5
g	4		

(b) sc

Id	min-smcs	Id	min-smcs
a	NULL	h	NULL
b	NULL	i	NULL
c	NULL	j	NULL
d	NULL	k	NULL
e	NULL	l	NULL
f	NULL	m	NULL
g	NULL		

(c) Initial state

Id	min-smcs	Id	min-smcs
a	G_1	h	G_1
b	G_2	i	NULL
c	G_2	j	NULL
d	NULL	k	NULL
e	G_2	l	NULL
f	G_2	m	NULL
g	G_1		

(d) After processing

Fig. 4. An example for the cache-based algorithm.

a) extract all nodes in H_u with Steiner-connectivity the same as u and whose minimal SMCSs have not been computed yet, stored into the set $S(H_u)$ (line 11);

b) invoke `Find-Minimal` procedure to compute minimal SMCSs for all nodes in $S(H_u)$ (line 12).

Here, the threshold η is used to balance the query time and the effectiveness of caching. The value of η is empirically tuned. Compared with previous algorithms on processing a query on node u, after computing the minimal SMCS $G[H_u]$, Algorithm 6 further computes a minimal SMCS for each node $v \in H_u$ with $sc(v) = sc(u)$ and does not have a minimal SMCS yet (all these nodes are stored into a set $S(H_u)$). Although $G[H_u]$ is a minimal SMCS of u, it might not be a minimal SMCS for each node $v \in S(H_u)$, which means a further refinement step is needed. However, since $G[H_u]$ is expected to be small, the refinement is expected to be much faster.

Example 6.1. As shown in Fig. 4a, suppose the query $Q = \{a\}$ and it is the first query (all pointers are initialized to NULL as shown in Fig. 4c). Algorithm 6 will return G_1 as a minimal SMCS of node a and set `min-smcs(a)` to G_1. Assuming that the threshold $\eta \geq |H_u|$, then we select all nodes in $V(G_1)$ with the Steiner-connectivity being 4 and `min-smcs` being NULL, i.e., $S(H_u) = \{b, c, e, f, g, h\}$. Here the Steiner-connectivities of all nodes are shown in Fig. 4b. Finally, after calling `Find-Minimal`$(G_1, S(H_u), V(G_1),$ `4)`, the minimal SMCSs for all nodes in $S(H_u)$ will be computed, as shown in Fig. 4d.

Suppose that the average query time (lines 4-12) is t_{total}, the average computation time of the cache part (lines 11-12) is t_{cache} and the average number of nodes cached during each query is $\mathcal{N}_{cached}$ (the average size of $S(H_u)$). In our experiments, the extra cost for caching, t_{cache}, is a negligible portion of the total query time t_{total}. Moreover, the amortized time $t_{amort} = \frac{t_{total}}{\mathcal{N}_{cached}}$, which is the expected average query time for future queries, is much smaller than the average query time of those algorithms without caching.

In the following, we will describe the `Find-Minimal` procedure in detail. First, in Section 6.1.1, we introduce the main component of the `Find-Minimal` procedure, named `Test-Minimal`, which is used for testing the minimality for a group of nodes. Based on that, we illustrate a subtle algorithm, `Find-Minimal`, which can find a minimal SMCS for each node in a group of nodes in Section 6.1.2.

6.1.1 Testing Minimality for a Group of Nodes

Given an SMCS $G[H]$ of a node $u \in H$ such that $sc(u) = k$, to test whether $G[H]$ is a minimal SMCS of u, we need to either (1) find a k-connected proper induced subgraph of $G[H]$ containing u, or (2) show that any proper induced subgraph of $G[H]$ containing u is not k-connected. In the second case, for each node $v \in H$, we need to test its separability, i.e., compute the k-components of the graph obtained by removing v. Hence testing minimality for an SMCS $G[H]$ of a node u requires at least $\Omega(|H| \cdot \mathsf{M})$ time.

Algorithm 7. `Test-Minimal`$(G[H], S, T, k)$

```
Input: A graph $G[H]$, a set $S \subseteq H$, a set $T$ of nodes whose
       separabilities are not tested, $k = \lambda(G[H])$
Output: A set KCC of $k$-connected node disjoint subgraphs
        of $G[H]$ containing some nodes in $S$ and a set $T$ of
        nodes whose separabilities are not tested
1  while $T \neq \emptyset$ do
2      pick a node $u \in T, T \leftarrow T \backslash u$;
3      KCC $\leftarrow$ getComponents$(G[H] \backslash u, k, S)$;
4      if KCC $\neq \emptyset$ then
5          return (KCC, $T$);
6  return $(\emptyset, \emptyset)$;
7  procedure getComponents$(G, k, S)$
8      $\phi_k \leftarrow$ ComputeKECCs$(G, k)$;
9      KCC $\leftarrow \emptyset$;
10     for each $k$-component $G[C] \in \phi_k$ do
11         if $C \cap S \neq \emptyset$ then
12             KCC $\leftarrow$ KCC $\cup \{C\}$;
13     return KCC
```

To speed up the computation, we do the minimality testing for a group of nodes S with the same Steiner-connectivity simultaneously as follows. We outline the minimality testing algorithm in Algorithm 7. Specifically, let T be the set of nodes in H whose separabilities are not tested. We repeatedly pick a node $u \in T$ (line 2) and test its separability for all nodes in S simultaneously by computing the k-components of $G[H] \backslash u$ that contains at least one node in S (line 3). If any k-component $G[C]$ is returned, then we know that u is separable for all nodes in $C \cap S$ and hence $G[H]$ is not a minimal SMCS for nodes in $C \cap S$. Otherwise, we can conclude that u is non-separable for all nodes in $S \backslash u$.

Lemma 6.1. *Given a graph $G[H]$ and a set $S \subseteq H$ of nodes such that for all $u \in S$, $sc(u) = \lambda(G[H]) = k$,* `Test-Minimal` *$(G[H], S, H, k)$ returns:*

- *$(\emptyset, \emptyset)$ if $G[H]$ is a minimal SMCS for all $v \in S$; or*
- *(KCC, T), where KCC is the k-connected node disjoint subgraphs of $G[H]$ containing some nodes in S and T is the nodes whose separabilities are not tested.*

The algorithm runs in $O(t \cdot \mathsf{M})$ time, where $t = |H \backslash T|$.

Proof. First, if for each node $v \in S$, $G[H]$ is a minimal SMCS of v, then there will be no k-component containing v in the graph obtained by removing any arbitrary node from $G[H]$ (otherwise $G[H]$ is not minimal for v). Hence we know that in this case, in `Test-Minimal`$(G[H], S, H, k)$, all nodes in H must be non-separable and $(\emptyset, \emptyset)$ will be returned.

Now suppose that there exists $v \in S$ such that $G[H]$ is not a minimal SMCS of v. Let G_v, which must be k-connected, be the minimal SMCS of v such that G_v is a proper induced subgraph of $G[H]$. Then all nodes $w \in H \backslash V(G_v)$

are separable for v. Hence when $w \in H \backslash V(G_v)$ is picked, the k-component containing v must be contained in KCC because it is k-connected and contains $v \in S$.

Since one node in T will be removed in each iteration and the running time of each iteration is bounded by $O(\mathsf{M})$, if t nodes are removed from T eventually, then the total running time of the algorithm is upper bounded by $O(t \cdot \mathsf{M})$. □

6.1.2 Finding Minimal SMCSs for a Group of Nodes

Based on Algorithm 7 for testing minimality, Algorithm 8 computes a pointer `min-smcs`(u) for each node $u \in S$ that points to a minimal SMCS of u, where T is a set of nodes in H whose separabilities are not tested. All pointers are initialized NULL. At first, we invoke `Test-Minimal` to do minimality testing (line 1). If $G[H]$ passes the minimality testing, we set `min-smcs`(u) to $G[H]$ for all $u \in S$ (lines 2-4). Otherwise, $G[H]$ is not minimal SMCS for some $u \in S$, and a collection KCC of k-components will be returned. Hence we can recursively compute the minimal SMCS within each component in KCC (lines 6-8). If not all nodes of S are contained in components of KCC, then we need to compute the minimal SMCSs within $G[H]$ for those nodes (line 10). However, since all previously sampled nodes in `Test-Minimal` are non-separable for them, we only need to sample from the nodes T' that has not been sampled yet.

Lemma 6.2. *At the end of* `Find-Minimal`$(G[H], S, H, k)$, *for all* $u \in S$, `min-smcs`(u) *points to a minimal SMCS of* u. *The algorithm runs in* $O(c \cdot |H| \cdot \mathsf{M})$ *time, where* c *is the maximum depth of recursive calls of* `Find-Minimal` *in line 7.*

Proof. *Correctness*. First it is easy to observe that whenever we set `min-smcs`(u) to $G[H]$ for $u \in S$, we must have $(\emptyset, \emptyset) =$ `Test-Minimal`$(G[H], S, T, k)$, which by Lemma 6.1 means that $G[H]$ is a minimal SMCS of u.

Running Time. Note that the main running time of `Find-Minimal` comes from the calls of `ComputeKECCs`, which can be done in $O(\mathsf{M})$ time. Since each call of the function `ComputeKECCs` is associated with a node u picked from T in `Test-Minimal`, we can charge the running time M of `ComputeKECCs`$(G[H] \backslash u, k, S)$ to node u.

Since in each level of recursive call of `Find-Minimal`, each node in H will be charged at most once, we conclude that the total running time of Algorithm 8 is upper bounded by $O(c \cdot |H| \cdot \mathsf{M})$, where c is the maximum depth of recursive calls of `Find-Minimal` in line 7. □

Algorithm 8. `Find-Minimal`$(G[H], S, T, k)$

```
1  $(\mathsf{KCC}, T') \leftarrow$ Test-Minimal$(G[H], S, T, k)$;
2  if $\mathsf{KCC} = \emptyset$ then
3      for each $u \in S$ do
4          min-smcs$(u) \leftarrow G[H]$;
5  else
6      for each $C \in \mathsf{KCC}$ do
7          Find-Minimal$(G[C], S \cap C, T' \cap C, k)$;
8          $S \leftarrow S \backslash C$;
9      if $S \neq \emptyset$ then
10         Find-Minimal$(G[H], S, T', k)$;
```

6.1.3 Incremental Removal Optimization

The previous idea of incremental sampling in Algorithm 4 can also be applied to the computation of minimal SMCSs for a group of nodes. In Algorithm 8, whenever we sample a node $u \in T$ and compute KCC of $G[H] \backslash u$, u is separable for all nodes in S that are contained in components of KCC, which means that we can increment the sample size by 1 in the recursive calls on components of KCC (line 7). For example, if we sample i nodes from T and $\mathsf{KCC} \neq \emptyset$ is the k-components containing nodes in S, then in each recursive call on component in KCC, $i+1$ nodes will be sampled. For the nodes of S not contained in KCC, we reset the sample size to 1 in the recursive call in line 10, while keeping those i sampled nodes in T.

6.2 Cache-Based Approximation Processing

Based on our previous algorithm (Algorithm 5) for computing an approximation of minimal SMCS, we consider in this section how to cache on approximate minimal SMCS solutions. In the following, we fix some global variables that can be accessed by all algorithms: let r be the approximation ratio and δ be the failure probability; the pointer `approx-min-smcs` is used to record the approximate minimal SMCS, instead of `min-smcs` used in Section 6.1.

Algorithm 9 takes a graph $G[H]$, a set $S \subseteq H$ of nodes with the same Steiner-connectivity k, a set $T \subseteq H$ of nodes to be sampled, the Steiner-connectivity k and the number *step* of consecutive successful tests as the input, returns (1) $(\emptyset, \emptyset)$ if for each $v \in S$, with probability $(1-\delta)$, $G[H]$ is an r-approximation of a minimal SMCS of v; or (2) (KCC, T), where KCC is a collection of k-components such that each of them contains some node in S, and T is the set of nodes not sampled yet.

Given Algorithm 9 for testing approximate minimal SMCS, Algorithm 10 finds an r-approximation of minimal SMCS for each node with probability at least $(1-\delta)$.

Algorithm 9. `Approx-Test-Minimal`$(G[H], S, T, k, step)$

```
1  while $T \neq \emptyset$ and $step < \omega$ do
2      sample a node $u \in T$ uniformly at random, $T \leftarrow T \backslash u$;
3      $\mathsf{KCC} \leftarrow$ getComponents$(G[H] \backslash u, k, S)$;
4      if $\mathsf{KCC} \neq \emptyset$ then
5          return $(\mathsf{KCC}, T, step)$;
6      else
7          $step \leftarrow step + 1$;
8  return $(\emptyset, \emptyset, step)$;
```

Algorithm 10. `Approx-Find-Minimal`$(G[H], S, T, k, step)$

```
1  $(\mathsf{KCC}, T', step') \leftarrow$
   Approx-Test-Minimal$(G[H], S, T, k, step)$;
2  if $\mathsf{KCC} = \emptyset$ then
3      for each $u \in S$ do
4          approx-min-smcs$(u) \leftarrow G[H]$;
5  else
6      for each $C \in \mathsf{KCC}$ do
7          Approx-Find-Minimal$(G[C], S \cap C, T' \cap C, k, 0)$;
8          $S \leftarrow S \backslash C$;
9      if $S \neq \emptyset$ then
10         Approx-Find-Minimal$(G[H], S, T', k, step')$;
```

Since whenever we set `approx-min-smcs`$(u) = G[H]$, we must have sampled ω consecutive nodes non-separable for u in $G[H]$, we have the following lemma.

TABLE 1
Dataset Statistics ($\mathbf{K} = 10^3$ and $\mathbf{M} = 10^6$)

	ID	Dataset	#Nodes	#Edges	$\overline{d}$	sc_{max}
Real	D1	ca-CondMat	21**K**	91**K**	8.6	25
	D2	soc-Epinions1	75**K**	405**K**	10.7	67
	D3	DBLP	803**K**	3.2**M**	8.2	118
	D4	wiki-Talk	2.3**M**	4.6**M**	3.9	131
	D5	as-Skitter	1.7**M**	11**M**	13.1	111
	D6	uk-2002	18**M**	261**M**	28.3	943
Synthetic	S1	Syn-1	20**K**	229**K**	22.9	94
	S2	Syn-2	40**K**	432**K**	21.6	100
	S3	Syn-3	60**K**	682**K**	22.8	124
	S4	Syn-4	80**K**	890**K**	22.2	118
	S5	Syn-5	100**K**	1.1**M**	22.4	134

Lemma 6.3. *When* `Approx-Find-Minimal`$(G[H], S, H, k, 0)$ *terminates, for each node $u \in S$, with probability $(1-\delta)$,* `approx-min-smcs`(u) *points to an r-approximation of a minimal SMCS of u.*

Proof. By Lemma 5.1, it suffices to prove that whenever we set `approx-min-smcs`$(u) = G[H]$, we must have sampled ω consecutive nodes non-separable for u in $G[H]$.

Notice that $step$ is the number of non-separable nodes that are consecutively sampled for each node $u \in S$, since the value of $step$ is increased by 1 if and only if a non-separable node for u is detected. Whenever we sampled a node separable for any $u \in S$, we find a k-component $C \in$ KCC containing u. When `Approx-Find-Minimal` $(G[C], S \cap C, T' \cap C, k, 0)$ is called (line 7), we reset $step$ to 0.

Hence when KCC $= \emptyset$, we can claim that either (1) $T = \emptyset$, which means $G[H]$ contains only non-separable nodes and is a minimal SMCS for all nodes in S; or (2) $step \geq \omega$, which means we must have consecutively sampled at least ω non-separable nodes for all nodes $u \in S$ in $G[H]$. □

7 Experimental Results

Data. We use six large real graph datasets: (1) *ca-CondMat*, or condensed matter collaboration network; (2) *soc-Epinions1*, the who-trusts-whom network of Epinions.com; (3) *DBLP*, a bibliographic network[1]; (4) *wiki-Talk*, a Wikipedia talk (communication) network; (5) *as-Skitter*, the Internet topology; and (6) *uk-2002*, the Web graph within the .uk domain in 2002.[2] Apart from DBLP and uk-2002, the datasets are downloaded from the Stanford SNAP library.[3] In addition, we generate 5 synthetic graphs using an open-sourced benchmark graph generator [25].[4]

For each dataset, we use its largest connected component as our test graph. Their number of nodes and edges, average degree $\overline{d}$, and the largest Steiner-connectivity sc_{max}, are reported in Table 1.

Queries. The node set Q of a query is randomly generated, based on query size $|Q|$ and the inter-distance l (i.e., the maximum distance between any two nodes in Q). For instance, a value of $l = 2$ means that all query nodes are within distance 2 to each other. By default, $|Q| = 3$ and $l = 2$. These values are also used in [24], [33]. For testing the effect of the Steiner-connectivity, we use a slightly different query model, as detailed in Section 7.2.

Algorithms. We tested several minimal SMCS solutions:

- `Basic`: Based on `Expand-Refine` (Algorithm 1), but compute maximum SMCS in line 2, and replace line 3 by `Refine-Basic` (Section 4.2.1).
- `ER`: Algorithm 1.
- `IncER`: Algorithm 1 with incremental removal optimization, i.e., replace line 3 by `Refine-Inc` (Algorithm 4).
- `AppIncER`$_\omega$: Approximate `Expand-Refine` with incremental removal optimization, i.e., in Algorithm 1, replace line 3 by `Approx-Refine-Inc` (Section 5.2). Here $\omega = \lceil \frac{\log \frac{1}{\delta}}{\log r} \rceil$ is the termination threshold, i.e., `AppIncER`$_\omega$ stops after ω non-separable nodes are sampled.
- `AppIncBER` $_\omega$: `AppIncER`$_\omega$ with bounded local search, i.e., in Algorithm 1, replace line 2 by `Bounded-Expand` (Section 5.1) and line 3 by `Approx-Refine-Inc` (Section 5.2). We set the local expansion threshold $\theta = 10{,}000$, which is selected to achieve stable quality and efficiency by testing θ in [1,000, 20,000] (Exp-6 in Section 7.3).

To examine the effectiveness of our solutions, we have also implemented the following algorithms:

- `max-SMCS`: the algorithm proposed in [6], which finds the maximum SMCS of Q.
- `local-SMCS`: this uses local expansion (Algorithm 2) to generate an SMCS of Q, without any refinement.
- `LCTC`: the algorithm proposed in [24], which finds the closest k-truss community of Q.
- `GrCon`: the algorithm proposed in [4],[5] which finds a k-core community of Q with small size.

We also study the cache-based algorithms:

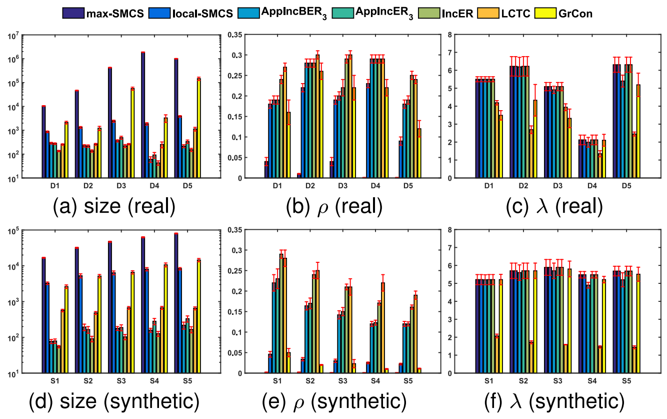
- `IncCache`: Algorithm 6 with incremental removal optimization, replace line 8 by `Refine-Inc` (Algorithm 4) and line 12 by `Find-Minimal` (Algorithm 8) with incremental removal optimization (Section 6.1.3).
- `AppIncCache` $_\omega$: Approximate cache-based algorithm with incremental removal optimization, i.e., in Algorithm 6, replace line 8 by `Approx-Refine-Inc` (Section 5.2) and line 12 by `Approx-Find-Minimal` (Algorithm 10).

For the parameter ω used in approximation algorithms, we found that ω = 3 balances the running time, approximation ratio and failure probability (details in Section 7.3). We thus set its default value to 3.

The above algorithms are implemented in C++ and compiled with GNU g++ 4.6.3 with the -O3 optimization except for `GrCon` which is implemented in Java. The source codes for computing k-components (`ComputeKECC`), constructing the connectivity graph, and computing the Steiner-connectivity and the maximum SMCS are obtained from the authors in [6], [8]. The source codes of both LCTC (binary version) and GrCon are obtained from the authors in [24] and [4] respectively. Our experiments are conducted on a machine with an Intel(R) Xeon (R) CPU@2.6 GHz and 96 GB memory running Linux.

Next, we examine the effectiveness and efficiency of SMCS solutions in Sections 7.1 and 7.2 respectively. We discuss the results for our relaxation solutions in Section 7.3.

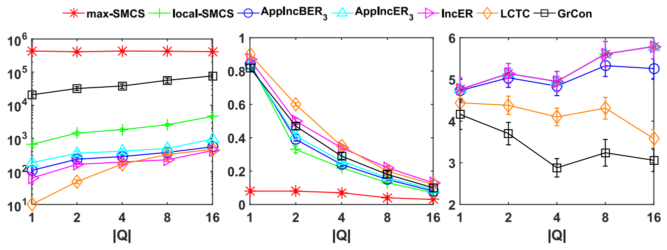
1. http://dblp.uni-trier.de/xml/
2. http://law.di.unimi.it/datasets.php
3. http://snap.stanford.edu/data/
4. http://santo.fortunato.googlepages.com/benchmark.tgz
5. http://bit.ly/1b6WbSQ



(a) size (real) (b) ρ (real) (c) λ (real)

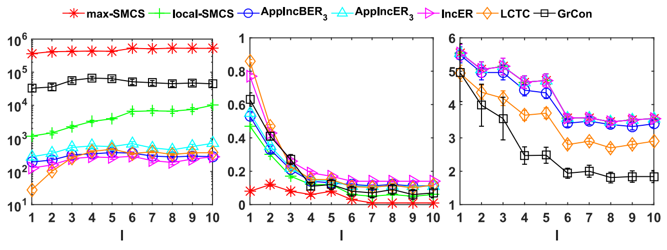
(d) size (synthetic) (e) ρ (synthetic) (f) λ (synthetic)

Fig. 5. Quality evaluation on both real and synthetic graphs.



(a) size (b) ρ (c) λ

Fig. 6. Quality evaluation on DBLP by varying query size $|Q|$ ($l = 2$).



(a) size (b) ρ (c) λ

Fig. 7. Quality evaluation on DBLP by varying inter-distance l ($|Q| = 3$).

7.1 Effectiveness

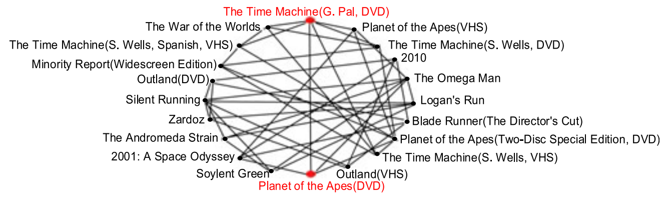
We compare the minimal SMCSs and approximate ones returned by `IncER`, `AppIncER`$_3$ and `AppIncBER`$_3$ respectively. We also evaluate the quality of `max-SMCS`, `local-SMCS`, `LCTC` and `GrCon` according to:

(1) Size. The number of nodes in the result graph.
(2) EdgeDensity ρ. This measures the density of a graph [20], [31], and is the ratio of the number of edges of a graph g to that of a complete graph with the same set of nodes: $\rho(g) = 2|E(g)|/(|V(g)| \times (|V(g)| - 1))$.
(3) Connectivity λ. The edge connectivity of the result graph (Definition 3.1).

Exp-1: Quality Evaluation. For each dataset in D1-D5 and S1-S5, we randomly select 500 sets of query nodes with $|Q|$ randomly ranging from 1 to 16 and inter-distance l being 2, and report the average values of size, edge density ρ and connectivity λ.

The results are shown in Fig. 5. Under the SMCS model, note that solutions with local expansion (`local-SMCS`, `AppIncBER`$_3$, `AppIncER`$_3$ and `IncER`) perform better than `max-SMCS` in terms of the result size and edge density. A gigantic number of nodes is returned by `max-SMCS`. Moreover, `IncER` achieves the highest edge density with the smallest number of nodes. Thus, it is useful to remove nodes from discovered subgraphs by our solution. The number of nodes in the subgraphs returned by `local-SMCS` is around 10 times more than `IncER`, which shows that using local expansion alone is not enough, and the refinement step is necessary. Compared with other community models, we can observe that 1) `IncER` performs much better than the k-core model (`GrCon`) under all measures; 2) even though the closest k-truss model (`LCTC`) can also return communities with small size and high edge density, `IncER` can return communities with higher connectivity. Since connectivity is a better cohesiveness metric as discussed in Section 2, the communities returned by `IncER` are more cohesive than those returned by `LCTC` and `GrCon`.

We also evaluate the effectiveness of our solutions using different kinds of queries on D3 (DBLP), i.e., by varying the query size $|Q|$ from 1 to 16, and the inter-distance l from 1 to 10. For each kind of queries, we randomly generate 500 sets of query nodes and report the average values of aforementioned measures. The results of varying the query size $|Q|$ (resp. the inter-distance l) are shown in Fig. 6 (resp. Fig. 7). In Figs. 6c and 7c, since `max-SMCS`, `local-SMCS`, `AppIncER`$_3$ and `IncER` all will find subgraphs with same connectivity, the corresponding lines overlap. We can observe that in



The Time Machine(G. Pal, DVD)
The War of the Worlds
Planet of the Apes(VHS)
The Time Machine(S. Wells, Spanish, VHS)
The Time Machine(S. Wells, DVD)
2010
Minority Report(Widescreen Edition)
Outland(DVD)
The Omega Man
Silent Running
Logan's Run
Zardoz
Blade Runner(The Director's Cut)
The Andromeda Strain
Planet of the Apes(Two-Disc Special Edition, DVD)
2001: A Space Odyssey
The Time Machine(S. Wells, VHS)
Soylent Green
Outland(VHS)
Planet of the Apes(DVD)

Fig. 8. A minimal SMCS of the Amazon network using query Q = {"The Time Machine (G. Pal, DVD)", "Planet of the Apes (DVD)"}.

TABLE 2
Quality Measures of the DBLP Case Study with Query Q = {"Michael Stonebraker", "Samuel Madden", "Daniel J. Abadi", "Jennie Duggan"}

Quality Metric	max-SMCS	local-SMCS	AppIncBER$_3$	AppIncER$_3$	IncER	LCTC	GrCon
size	171,435	129	17	17	15	83	211
ρ	0.0001	0.21	0.54	0.54	0.58	0.33	0.09
λ	7	7	7	7	7	7	7

TABLE 3
Quality Measures of the DBLP Case Study with Single-Node Query Q = {"Christian S. Jensen"}

Quality Metric	max-SMCS	local-SMCS	AppIncBER$_3$	AppIncER$_3$	IncER	LCTC	GrCon
size	10,265	39	21	21	21	21	12,423
ρ	0.004	0.59	1	1	1	1	0.003
λ	20	20	20	20	20	20	2

TABLE 4
Quality Measures of the Amazon Case Study with Query Q = {"The Time Machine (DVD)", "Planet of the Apes (DVD)"}

Quality Metric	max-SMCS	local-SMCS	AppIncBER$_3$	AppIncER$_3$	IncER	LCTC	GrCon
size	122,219	148	23	23	20	2	143,562
ρ	0.0001	0.05	0.23	0.23	0.24	1	0.00001
λ	4	4	4	4	4	1	1

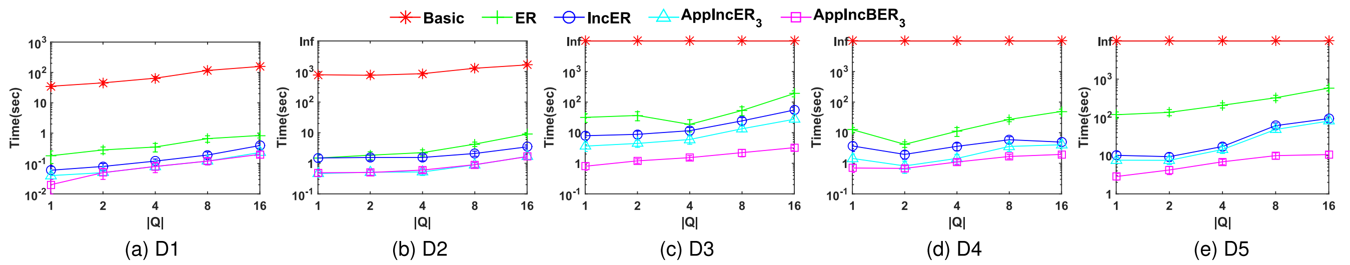


Fig. 9. Varying query size $|Q|$ ($l = 2$).

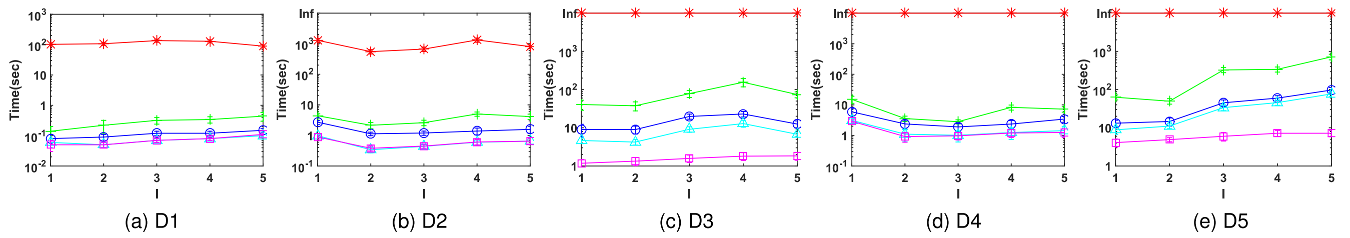


Fig. 10. Varying inter-distance l ($|Q| = 3$).

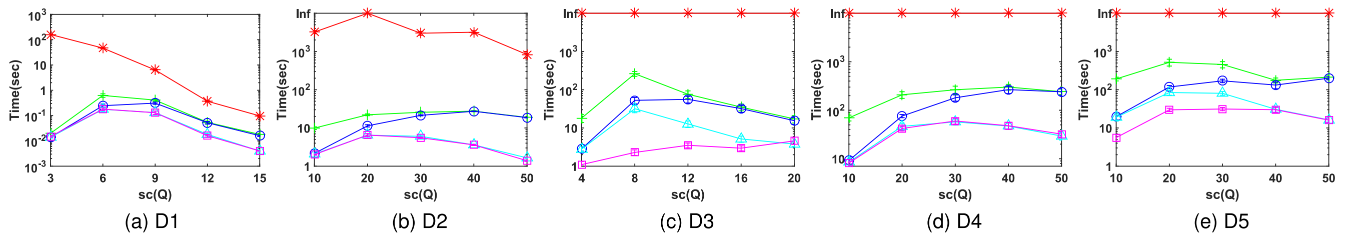


Fig. 11. Varying Steiner-connectivity of Q ($|Q| = 3$, $l = 2$).

general the size of results will increase when increasing $|Q|$ and l, while the edge density will decrease. Moreover, as shown in Fig. 6c, when $|Q|$ increases, only our SMCS model can achieve higher connectivity. As shown in Fig. 7c, when l increases, the connectivity will decrease for all methods since query nodes are farther. Similar to the result obtained from Fig. 5, compared with other competitors, our solutions can always return communities with small size, high edge density and high connectivity.

Exp-2: Case Study. We use the query Q = {"Michael Stonebraker", "Samuel Madden", "Daniel J. Abadi", "Jennie Duggan"} on the DBLP dataset. Table 2 shows the quality measures for different SMCS methods and two other community models. We found that IncER returns a small and cohesive 7-connected subgraph (size = 15 and ρ = 0.58), which is much better than those returned by max-SMCS, local-SMCS, LCTC and GrCon. Fig. 1 (in Section 1) shows the result returned by IncER. All authors found in the result for this query are established researchers in databases, and they indeed collaborate with each other frequently. Due to the large sizes of the subgraphs returned by max-SMCS, local-SMCS, LCTC and GrCon, they are not illustrated here. Notice that the results of both AppIncER$_3$ and AppIncBER$_3$ are all very close to the one returned by IncER.

In addition, for a single-node query Q = {"Christian S. Jensen"} on the DBLP dataset, the quality measures of the results returned by different methods are shown in Table 3. Our solutions and LCTC can find a 20-connected clique for "Christian S. Jensen", which is much better than those returned by max-SMCS, local-SMCS and GrCon.

Moreover, we also conduct a case study on an Amazon co-purchasing network[6] with 548K nodes and 1.78M edges. Each node denotes a product, and an edge between two products indicates that they have been purchased together. The quality measures of the results returned by different methods for the query Q = {"The Time Machine (G. Pal, DVD)", "Planet of the Apes (DVD)"} are shown in Table 4. IncER returns a small and cohesive 4-connected subgraph (size = 20 and ρ = 0.24) which is much better than those returned by max-SMCS, local-SMCS, LCTC and GrCon. Specifically, LCTC cannot find any other nodes since trussness is too restrictive on the triangle structure. GrCon returns a very large community with 143,562 nodes which is not useful in practice. The result returned by IncER is shown in Fig. 8. All products found in the result for this query are science fiction movies (DVD or VHS type), and are highly connected to each other.

7.2 Efficiency

We now evaluate the efficiency of our algorithms for minimal SMCS queries under different situations. Each experiment is run three times, and the average CPU time is reported in seconds. In the experiments, we have not reported the offline indexing time for computing the connectivity graph and its maximum spanning tree (MST), since they are already reported in [6], e.g., 141 seconds for D5 (as-Skitter). We treat the running time of a query as infinite (Inf) if it exceeds 1 hour.

6. http://snap.stanford.edu/data/amazon-meta.html

TABLE 5
Efficiency Comparison on DBLP (in Seconds)

$\vert Q\vert$	1	2	4	8	16
LCTC	0.3	3.6	3.7	4.0	4.3
AppIncBER$_3$	0.8	1.2	1.5	2.2	3.2
l	1	2	3	4	5
LCTC	3.6	3.7	3.8	3.9	3.9
AppIncBER$_3$	1.2	1.3	1.6	1.8	1.8

TABLE 6
Scalability Testing for AppIncBER$_3$ on D6 (in Seconds)

$\vert Q\vert$	1	2	4	8	16
Time	13.1 ± 0.9	21.3 ± 1.2	28.2 ± 1.5	32.5 ± 1.9	37.9 ± 2.1
l	1	2	3	4	5
Time	34.3 ± 1.8	41.4 ± 2.3	27.5 ± 1.6	32.4 ± 1.7	29.6 ± 1.5
$sc(Q)$	10	20	30	40	50
Time	19.1 ± 1.3	17.1 ± 1.0	16.1 ± 1.6	18.6 ± 2.7	18.4 ± 2.3

Exp-3: Effect of Queries. In these experiments, we test our approaches using different queries. The reported time is the average time of processing 500 queries.

First, we observe the effect of the query size $|Q|$. We test five different $|Q|$ values in {1, 2, 4, 8, 16}. The running time of ER, IncER, AppIncER$_3$, AppIncBER$_3$ and Basic on different datasets (D1-D5) is shown in Fig. 9. In general, the running time of all algorithms increases with the query size. Since the number of nodes in the candidate generated in the local expansion step increases when $|Q|$ increases, the time cost in both candidate generation step and refinement step increases. Our algorithms (ER, IncER, AppIncER$_3$ and AppIncBER$_3$) outperform the baseline algorithm by several orders of magnitude on all datasets. Moreover, IncER is better than ER. For example, on D5 (as-Skitter), IncER is around 10 times faster than ER. Thus, the incremental removal optimization improves the performance substantially. The approximation algorithm AppIncER$_3$ further improves the performance by relaxing the minimality; on D3 (DBLP), AppIncER$_3$ is around 2 times faster than IncER. AppIncBER$_3$ achieves the highest performance by relaxing both connectivity and minimality; on D3 (DBLP), AppIncBER$_3$ is around 10 times faster than IncER.

We then study the effect of the inter-distance l among query nodes. The running time of ER, IncER, AppIncER$_3$, AppIncBER$_3$, and Basic on different datasets (D1-D5) by varying the inter-distance l (from 1 to 5) of query is illustrated in Fig. 10. Similar to the results obtained by varying the query size, ER, IncER, AppIncER$_3$ and AppIncBER$_3$ outperform Basic by several orders of magnitude on all datasets. IncER is still better than ER. For example, on D5 (as-Skitter), IncER is around 5.7 times faster than ER. Again, AppIncBER$_3$ performs the best; on D5, AppIncBER$_3$ is around 45 times faster than ER.

We also evaluate the effect of the Steiner-connectivity $sc(Q)$ of Q. For each dataset in D1-D5, we select some representative values of Steiner-connectivity based on its largest Steiner-connectivity. For each value of $sc(Q)$, we randomly select 500 different query sets with $|Q| = 3$ and $l = 2$ as follows. First, we compute the connectivity graph of G, as well as sc values for each node in G. Second, we randomly select a node q from the nodes with $sc() = k$, and use Breadth First Search (BFS) to find a candidate node set, C. During the

TABLE 7
Scalability Testing for AppIncBER$_3$ on D6 with $|Q| = 16$ and $l = 5$ (in Seconds)

$sc(Q)$	10	20	30	40	50
Time	36.1 ± 2.3	32.5 ± 2.5	29.3 ± 3.3	31.7 ± 4.1	37.9 ± 5.2

procedure of BFS, assuming that the node being visited is u, an edge (u, v) will be visited **iff** the distance between q and v is less than $l/2$ and $sc(\{u, v\}) \geq k$. Finally, if we can select $|Q| - 1$ nodes, say Q' from the candidate set $C \backslash q$, then the query set $Q = q \cup Q'$; otherwise repeat the second step until a valid query set is generated. The running time of ER, IncER, AppIncER$_3$, AppIncBER$_3$ and Basic on different datasets by varying the Steiner-connectivity $sc(Q)$ of query is illustrated in Fig. 11. Generally, ER, IncER, AppIncER$_3$ and AppIncBER$_3$ outperform the baseline algorithm by several orders of magnitude on all datasets. When $sc(Q)$ increases, the advantage of the approximation algorithm over others becomes more obvious. For example, on D5 (as-Skitter), when $sc(Q)$ = 50, AppIncER$_3$ is around 12.5 times faster than IncER. When $sc(Q)$ increases, the number of nodes in the result increases, and the approximation algorithm stops earlier while the exact one needs a lot of time to do minimality testing on the result graph. AppIncBER$_3$ which combines both connectivity and minimality relaxations still is the fastest.

Finally, we report the efficiency comparison between our solution (AppIncBER$_3$) and the best competitor LCTC on the DBLP dataset by varying $|Q|$ from 1 to 16 and l from 1 to 5. As shown in Table 5, our solution is around twice faster than LCTC in almost all cases.

Exp-4: Scalability Testing. We test the scalability of our fastest algorithm (AppIncBER$_3$) on the largest dataset D6 (uk-2002) which contains 261 million edges. The reported time is the average time of processing 500 queries. Table 6 shows the running time of AppIncBER$_3$ on D6 when varying the query size $|Q|$, inter-distance l and Steiner-connectivity $sc(Q)$. It shows that AppIncBER$_3$ has ideal scalability and is quite efficient even for such a large network.

We also conduct experiments by varying Steiner-connectivity $sc(Q)$ with large $|Q|$ (=16) and l (=5). The result is reported in Table 7. Similar to the results obtained in Table 6, AppIncBER$_3$ can handle such kinds of queries efficiently in such a large dataset.

Exp-5: Cache-Based Algorithms. To evaluate the cache-based algorithms for single-node queries, we compare the exact algorithm IncCache and the approximate version AppIncCache$_3$ with two other methods without caching, IncER and AppIncER$_3$, respectively. We set the parameter η to 10,000, which is selected to balance the query processing time and the effectiveness of caching by testing $\eta \in [1{,}000, 20{,}000]$. We conduct experiments as follows. 10K single-node queries ($|Q| = 1$) are generated randomly for each dataset. We report the running time of the four algorithms in Table 8. For IncCache and AppIncCache$_3$, we report the average query time t_{total}, the average time t_{cache} for caching, the amortized time t_{amort}, and the additional space cost for caching. For both IncCache and AppIncCache$_3$, the average time t_{cache} for caching does not dominate the query time t_{total}. For instance, on D3 (DBLP), the time t_{cache} for caching of IncCache (AppIncCache$_3$) accounts for 10 percent (20 percent) of the whole query time. Although the query time of IncCache and

TABLE 8
Single-Node Query Time (in Seconds)

	Exact algorithm					Approximation algorithm				
		IncCache					AppIncCache$_3$			
Graph	IncER	t_{cache}	t_{total}	t_{amort}	Space	AppIncER$_3$	t_{cache}	t_{total}	t_{amort}	Space
D1	0.1 ± 0.02	0.001 ± 0.0001	0.02 ± 0.002	**0.016**	1 MB	0.04 ± 0.01	0.01 ± 0.001	0.02 ± 0.0001	**0.02**	2 MB
D2	1.5 ± 0.4	0.4 ± 0.1	1.3 ± 0.1	**1.0**	3 MB	0.5 ± 0.1	0.1 ± 0.04	0.5 ± 0.03	**0.4**	6 MB
D3	8.0 ± 2.5	0.7 ± 0.1	6.7 ± 0.5	**1.5**	14 MB	3.6 ± 1.2	0.8 ± 0.2	3.8 ± 0.2	**0.8**	37 MB
D4	3.7 ± 1.3	0.5 ± 0.1	2.4 ± 0.3	**2.2**	26 MB	1.4 ± 0.6	0.2 ± 0.1	1.0 ± 0.1	**0.9**	27 MB
D5	10.6 ± 2.5	1.5 ± 0.2	12.0 ± 1.0	**3.4**	22 MB	7.6 ± 1.4	2.0 ± 0.4	8.9 ± 0.4	**2.2**	41 MB

AppIncCache$_3$ is close to IncER and AppIncER$_3$ respectively, their amortized time t_{amort}, which is the expected average query time for future queries, is much smaller than IncER and AppIncER$_3$. For example, on D3 (DBLP), the amortized time t_{amort} of IncCache (AppIncCache$_3$) is 5.4 (4.6) times faster than the query time of IncER (AppIncER$_3$). Moreover, the space cost is relatively very small for caching.

We also conduct experiments by varying the number of single-node queries from 1K to 10K. The results on the DBLP dataset are shown in Table 9. We can observe that the time cost is stable when the number of queries increases. Meanwhile, the space cost for caching will increase because more results are cached. Nevertheless, the space cost is relatively very small.

7.3 Results on Constraint Relaxation

In what follows, we will first evaluate the error of connectivity by adding a threshold in the *Expand* step. Then we evaluate our approximation techniques in the *Refine* step and discuss how to set parameters.

Exp-6: Connectivity Relaxation. We evaluate the sensitivity of the local expansion threshold θ and the error of connectivity in AppIncBER$_3$ on D3 (DBLP). To evaluate the sensitivity of θ, for each θ in {1K, 5K, 10K, 15K, 20K}, we randomly select 500 different query sets with $|Q|$ ranging from 1 to 16 and the inter-distance being 2. The running time and the average percentage errors of the connectivity (k') of detected minimal SMCSs by AppIncBER$_3$ to the one (k) returned by exact algorithms, e.g., ER, are reported in Fig. 12 ($\%_{\text{error}} = (k - k')/k$). Both of the running time and the average percentage errors increase when θ increases. We observe that when θ equals to 10K, AppIncBER$_3$ can achieve the best balance between the running time and the loss of connectivity. For the evaluation of the error of connectivity in AppIncBER$_3$, the average percentage errors are reported in Table 10, where we vary the query size $|Q|$, inter-distance l and Steiner-connectivity $sc(Q)$ as mentioned in Section 7.2. The connectivity of detected minimal SMCSs obtained by AppIncBER$_3$ are very close to the exact solutions. Combined with the analysis on efficiency in Section 7.2, AppIncBER$_3$ balances the efficiency and effectiveness well.

Exp-7: Minimality Relaxation. We evaluate the performance of the approximation algorithm (AppIncER$_\omega$) by varying the termination threshold ω, from 2 to 5. To investigate the actual approximation ratio and failure probability, for each value of ω from 2 to 5, we select a pair of (r, δ), i.e., $(8, 0.02)(\frac{\log\frac{1}{0.02}}{\log 8} \approx 1.88 < 2)$, $(5, 0.01)(\frac{\log\frac{1}{0.01}}{\log 5} \approx 2.86 < 3)$, $(6, 0.001)(\frac{\log\frac{1}{0.001}}{\log 6} \approx 3.86 < 4)$ and $(4, 0.001)(\frac{\log\frac{1}{0.001}}{\log 4} \approx 4.98 < 5)$. We randomly select 500 sets of query nodes with size randomly ranging from 1 to 16 and the inter-distance l being 2. The query time, actual approximation ratio and failure probability of AppIncER$_2$, AppIncER$_3$, AppIncER$_4$ and AppIncER$_5$ are shown in Fig. 13. In Fig. 13a, we also report

TABLE 9
Varying the Number of Single-Node Queries on DBLP (in Seconds)

	# Queries	1K	3K	5K	7K	10K
IncCache	t_{cache}	0.6 ± 0.2	0.7 ± 0.1	0.6 ± 0.1	0.6 ± 0.1	0.7 ± 0.1
	t_{total}	6.3 ± 1.5	6.3 ± 0.8	6.1 ± 0.6	6.0 ± 0.5	6.7 ± 0.5
	t_{amort}	1.2	1.2	1.3	1.3	1.5
	Space	8.6 MB	10.2 MB	11.6 MB	13 MB	14.4 MB
AppIncCache$_3$	t_{cache}	0.9 ± 0.4	1.3 ± 0.4	1.1 ± 0.2	1.0 ± 0.2	1.1 ± 0.2
	t_{total}	4.3 ± 0.7	4.1 ± 0.3	3.8 ± 0.2	3.7 ± 0.2	3.8 ± 0.2
	t_{amort}	0.7	0.7	0.7	0.7	0.8
	Space	10.8 MB	19 MB	23.7 MB	28.7 MB	37 MB

TABLE 10
Error of Connectivity on D3 (DBLP)

$\|Q\|$	1	2	4	8	16
$\%_{\text{error}}$	$1\% \pm 0.2\%$	$1\% \pm 0.2\%$	$2\% \pm 0.4\%$	$3\% \pm 0.5\%$	$6\% \pm 0.9\%$
l	1	2	3	4	5
$\%_{\text{error}}$	$1\% \pm 0.4\%$	$1\% \pm 0.5\%$	$2\% \pm 0.6\%$	$2\% \pm 0.7\%$	$2\% \pm 0.8\%$
$sc(Q)$	4	8	12	16	20
$\%_{\text{error}}$	$1\% \pm 0.5\%$	$4\% \pm 0.7\%$	$3\% \pm 0.7\%$	$1\% \pm 0.2\%$	0%

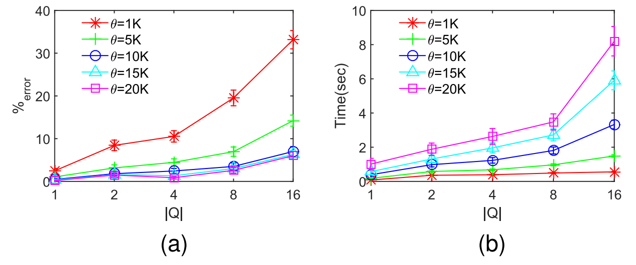


Fig. 12. Evaluation of θ on D3 (DBLP).

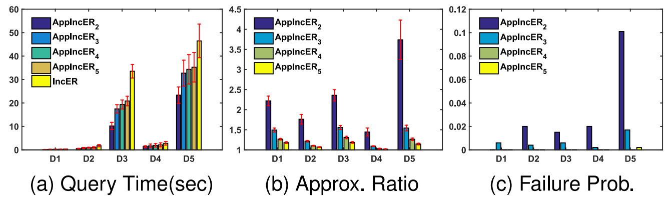


Fig. 13. Minimality approximation performance.

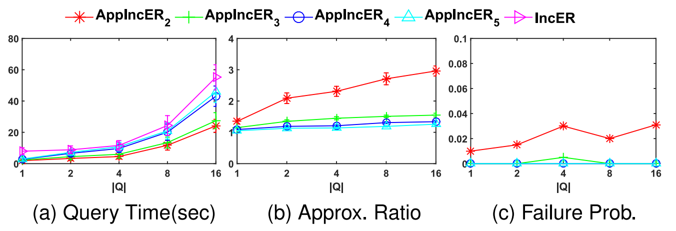


Fig. 14. Minimality approximation performance on DBLP by varying query size $|Q|$ ($l = 2$).

the query time of the exact algorithm IncER. In general, all these approximation algorithms run much faster than IncER, and their actual average approximation ratios and failure probabilities are all much lower than the theoretical values. One exception is that the failure probability of $\texttt{AppIncER}_2$ on D5 (as-Skitter) is much larger than its theoretical value, because the value of ω in $\texttt{AppIncER}_2$ is too small ($\omega = 2$) which leads to a high variance. Specifically, $\texttt{AppIncER}_2$ is the fastest approximation algorithm on all datasets, but its actual approximation ratio and failure probability are the highest. Although $\texttt{AppIncER}_5$ is the slowest, it achieves the best actual approximation ratio and failure probability. We observe that a larger ω leads to higher accuracy. We also see that $\texttt{AppIncER}_3$ achieves the best balance among the running time, approximation ratio, and failure probability.

Moreover, we also conduct experiments on D3 (DBLP) by varying the query size $|Q|$ from 1 to 16 with the inter-distance l being 2. For each kind of queries, we randomly generate 500 sets of query nodes and report the average values of query time, actual approximation ratio and failure probability for each solution. The results are shown in Fig. 14. Similar to the results obtained from Fig. 13, $\texttt{AppIncER}_3$ achieves the best balance. Hence, we thus suggest to set ω to 3 (with $r = 5$ and $\delta = 0.01$).

Summary. In our experiments, we have tested different kinds of query settings, i.e., the query size $|Q|$ varies from 1 to 16 and the inter-distance l varies from 1 to 10, which cover a wide range of common queries. Moreover, the average degree of real and synthetic datasets in our experiments varies from 3.9 to 28.3, which is common in a variety of graphs. All these experiments show that our model performs better than other competitors in terms of fewer nodes, and higher edge density and connectivity. In addition, the efficiency of our minimal SMCS solutions addresses several orders of magnitude improvement over basic solutions. The proposed cash-based solution for single-node SMCS queries demonstrates further improvement in speed. Finally, following the recommended parameter settings, the loss caused by connectivity relaxation and minimality relaxation is negligible.

8 Conclusions

In this paper, we examine the minimal SMCS problem. We develop Expand-Refine algorithms for finding minimal SMCSs. In addition, we propose two strategies to further improve the efficiency by relaxing connectivity and minimality. We also design fast cache-based processing techniques for single-node queries. Our experiments on large datasets demonstrate the effectiveness and efficiency of our proposed solutions. We plan to extend our techniques to cohesive subgraph search under other metrics.

In the future, we also plan to investigate how to extend cache-based solution to support multiple-node queries. Let Q be a multiple-node query with $|Q| > 1$. A naive extension is described as follows. After Q is initiated, we compute its minimal SMCS, say G', using our ER algorithm. Then, we extract all possible multiple-node sets from G', whose sc value is the same as that of Q. Next, we compute all minimal SMCSs for all extracted sets using this method. However, the number of sets with a specific sc value, i.e., $O(2^{|V(G')|})$, can be very large. Hence, the total time of computing the minimal SMCSs for these sets can be extremely expensive. A lot of space is also needed to store the results. An interesting problem is then to study which sets should be chosen, for which their minimal SMCSs are computed, so that they can have a higher chance to be used by multiple-node query requests started later. We consider this problem as a future work.

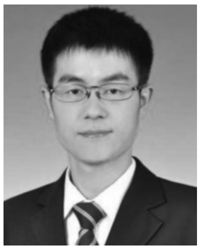
Acknowledgments

Reynold Cheng, Jiafeng Hu, Siqiang Luo, and Yixiang Fang were supported by the Research Grants Council of Hong Kong (RGC Projects HKU 17229116 and 17205115) and the University of Hong Kong (Projects 102009508 and 104004129). Xiaowei Wu was supported by the Hong Kong RGC grant 17217716. This research is conducted in part using the HKU ITS research computing facilities. The authors would like to thank the reviewers for their insightful comments.

References

[1] T. Akiba, Y. Iwata, and Y. Yoshida, “Linear-time enumeration of maximal K-edge-connected subgraphs in large networks by random contraction,” in *Proc. 22nd ACM Int. Conf. Inf. Knowl. Manage.*, 2013, pp. 909–918.
[2] O. Amini, D. Peleg, S. Pérennes, I. Sau, and S. Saurabh, “Degree-constrained subgraph problems: Hardness and approximation results,” in *Proc. Int. Workshop Approximation Online Algorithms*, 2008, pp. 29–42.
[3] S. Asthana, O. D. King, F. D. Gibbons, and F. P. Roth, “Predicting protein complex membership using probabilistic network reliability,” *Genome Res.*, vol. 14, pp. 1170–1175, 2004.
[4] N. Barbieri, F. Bonchi, E. Galimberti, and F. Gullo, “Efficient and effective community search,” *Data Mining Knowl. Discovery*, vol. 29, no. 5, pp. 1406–1433, Sep. 2015.
[5] M. G. Bell and Y. Iida, *Transportation Network Analysis*. Hoboken, NJ, USA: Wiley, 1997.
[6] L. Chang, X. Lin, L. Qin, J. X. Yu, and W. Zhang, “Index-based optimal algorithms for computing steiner components with maximum connectivity,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 459–474.
[7] L. Chang, J. Yu, and L. Qin, “Fast maximal cliques enumeration in sparse graphs,” *Algorithmica*, vol. 66, no. 1, pp. 173–186, 2013.
[8] 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. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 205–216.
[9] J. Cheng, Y. Ke, S. Chu, and M. T. Özsu, “Efficient core decomposition in massive networks,” in *Proc. IEEE 27th Int. Conf. Data Eng.*, 2011, pp. 51–62.
[10] 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. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 447–458.
[11] N. Chiba and T. Nishizeki, “Arboricity and subgraph listing algorithms,” *SIAM J. Comput.*, vol. 14, no. 1, pp. 210–223, 1985.
[12] A. Clauset, “Finding local community structure in networks,” *Phys. Rev. E*, vol. 72, Aug. 2005, Art. no. 026132.
[13] J. Cohen, “Trusses: Cohesive subgraphs for social network analysis,” Nat. Secur. Agency, Fort Meade, MD, USA, 2008, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.505.7006&rep=rep1&type=pdf

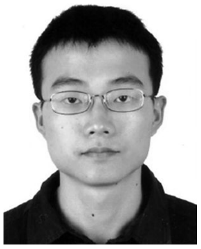
[14] C. J. Colbourn and C. Colbourn, *The Combinatorics of Network Reliability*. New York, NY, USA: Oxford Univ. Press, 1987.
[15] W. Cui, Y. Xiao, H. Wang, Y. Lu, and W. Wang, "Online search of overlapping communities," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 277–288.
[16] W. Cui, Y. Xiao, H. Wang, and W. Wang, "Local search of communities in large graphs," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2014, pp. 991–1002.
[17] Y. Fang, R. Cheng, X. Li, S. Luo, and J. Hu, "Effective community search over large spatial graphs," *Proc. VLDB Endowment*, vol. 10, no. 6, pp. 709–720, 2017.
[18] Y. Fang, R. Cheng, S. Luo, and J. Hu, "Effective community search for large attributed graphs," *Proc. VLDB Endowment*, vol. 9, no. 12, pp. 1233–1244, 2016.
[19] Y. Fang, R. Cheng, S. Luo, J. Hu, and K. Huang, "C-explorer: Browsing communities in large graphs," *Proc. VLDB Endowment*, vol. 10, no. 12, Aug. 2017.
[20] A. Gibbons, *Algorithmic Graph Theory*. Cambridge, U.K.: Cambridge Univ. Press, 1985.
[21] J. Hu, X. Wu, R. Cheng, S. Luo, and Y. Fang, "Querying minimal Steiner maximum-connected subgraphs in large graphs," in *Proc. 25th ACM Int. Conf. Inf. Knowl. Manage.*, 2016, pp. 1241–1250.
[22] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu, "Querying K-truss community in large and dynamic graphs," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2014, pp. 1311–1322.
[23] X. Huang and L. V. S. Lakshmanan, "Attribute-driven community search," *Proc. VLDB Endowment*, vol. 10, no. 9, pp. 949–960, May 2017.
[24] X. Huang, L. V. S. Lakshmanan, J. X. Yu, and H. Cheng, "Approximate closest community search in networks," *Proc. VLDB Endowment*, vol. 9, no. 4, pp. 276–287, Dec. 2015.
[25] A. Lancichinetti, S. Fortunato, and F. Radicchi, "Benchmark graphs for testing community detection algorithms," *Phys. Rev. E*, vol. 78, no. 4, 2008, Art. no. 046110.
[26] P. Lee and L. V. Lakshmanan, "Query-driven maximum quasi-clique search," in *Proc. SIAM Int. Conf. Data Mining*, 2016, pp. 522–530.
[27] R.-H. Li, L. Qin, J. X. Yu, and R. Mao, "Influential community search in large networks," *Proc. VLDB Endowment*, vol. 8, no. 5, pp. 509–520, Jan. 2015.
[28] R.-H. Li, J. Yu, and R. Mao, "Efficient core maintenance in large dynamic graphs," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 10, pp. 2453–2465, Oct. 2014.
[29] M. Maresca and H. Li, "Connection autonomy in SIMD computers: A VLSI implementation," *J. Parallel Distrib. Comput.*, vol. 7, no. 2, pp. 302–320, 1989.
[30] K. Mehlhorn, "A faster approximation algorithm for the Steiner problem in graphs," *Inf. Process. Lett.*, vol. 27, no. 3, pp. 125–128, 1988.
[31] L. Qin, R.-H. Li, L. Chang, and C. Zhang, "Locally densest subgraph discovery," in *Proc. 21th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2015, pp. 965–974.
[32] S. B. Seidman, "Network structure and minimum degree," *Social Netw.*, vol. 5, no. 3, pp. 269–287, 1983.
[33] M. Sozio and A. Gionis, "The community-search problem and how to plan a successful cocktail party," in *Proc. 16th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2010, pp. 939–948.
[34] V. Spirin and L. A. Mirny, "Protein complexes and functional modules in molecular networks," *Proc. Nat. Academy Sci. United States America*, vol. 100, no. 21, pp. 12123–12128, 2003.
[35] J. Wang and J. Cheng, "Truss decomposition in massive networks," *Proc. VLDB Endowment*, vol. 5, no. 9, pp. 812–823, 2012.
[36] S. Wasserman and K. Faust, *Social Network Analysis: Methods and Applications*. Cambridge, U.K.: Cambridge Univ. Press, 1994.
[37] D. R. White and F. Harary, "The cohesiveness of blocks in social networks: Node connectivity and conditional density," *Sociological Methodology*, vol. 31, no. 1, pp. 305–359, 2001.
[38] Y. Wu, R. Jin, J. Li, and X. Zhang, "Robust local community detection: On free rider effect and its elimination," *Proc. VLDB Endowment*, vol. 8, no. 7, pp. 798–809, 2015.
[39] 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.*, vol. 32, no. 2, 2007, Art. no. 13.



Jiafeng Hu received the BEng and ME degrees from Jilin University and the University of Chinese Academy of Sciences, in 2011 and 2014, respectively. He is currently working toward the PhD degree in the Department of Computer Science, University of Hong Kong (HKU), under the supervision of Dr. Reynold Cheng. His research interests include spatio-temporal data management and graph databases.

Xiaowei Wu received the BEng degree from the University of Science and Technology of China, in 2011 and the PhD degree in computer science from the University of Hong Kong, in 2015. He is currently a post-doctoral research fellow in the Department of Computer Science, The University of Hong Kong. His research interests include combinatorial optimization, approximation algorithms, and graph theory.

Reynold Cheng received the PhD degree from the Department of Computer Science, Purdue University, in 2005. He is an associate professor in the Department of Computer Science, University of Hong Kong. He received an Outstanding Young Researcher Award in 2011-2012 from HKU. He has served as a PC member and reviewer for international conferences (e.g., SIGMOD, VLDB, ICDE, and KDD) and journals (e.g., the *IEEE Transactions on Knowledge and Data Engineering*, the *ACM Transactions on Database Systems*, the *Very Large Data Base Journal*, and the *Information Systems*). He is an associate editor of the *IEEE Transactions on Knowledge and Data Engineering*, and was on the EIC selection committee of the *IEEE Transactions on Knowledge and Data Engineering*. He is a member of the IEEE.



Siqiang Luo received the BEng and MS degrees from Fudan University, in 2010 and 2013, respectively. He is working toward the PhD degree in the Department of Computer Science, University of Hong Kong (HKU) under the supervision of Prof. Benjamin C.M. Kao and Dr. Reynold Cheng. His research interests include the areas of data management and analytics.

Yixiang Fang received the BEng and MS degrees from the Harbin Engineering University and ShenZhen Graduate School, Harbin Institute of Technology, in 2010 and 2013, respectively. He is currently working toward the PhD degree in the Department of Computer Science, University of Hong Kong (HKU) under the supervision of Dr. Reynold Cheng. His research interests mainly focus on big data analytics on spatial-temporal databases, graph databases, uncertain databases, and web data mining.