Minimum Strongly Connected Subgraph Collection in Dynamic Graphs

Xin Chen  
The Chinese University of Hong Kong  
xchen@se.cuhk.edu.hk  

Jieming Shi*  
Hong Kong Polytechnic University  
jieming.shi@polyu.edu.hk  

You Peng  
The University of New South Wales  
unswpy@gmail.com  

Wenqing Lin  
Tencent  
edlin@tencent.com  

Sibo Wang  
The Chinese University of Hong Kong  
swang@se.cuhk.edu.hk  

Wenjie Zhang  
The University of New South Wales  
wenjie.zhang@unsw.edu.au  

ABSTRACT

Real-world directed graphs are dynamically changing, and it is important to identify and maintain the strong connectivity information between nodes, which is useful in numerous applications. Given an input graph $G$, we study a new problem, minimum strongly connected subgraph collection (MSCSC), which asks for a complete collection of subgraphs, each of which contains a maximal set of nodes that are strongly connected to each other via minimum number of edges in $G$.

MSCSC is NP-hard, and its computation and maintenance are challenging, especially on large-scale dynamic graphs. Thus, we resort to approximate MSCSC with theoretical guarantees. We develop a series of approximate MSCSC methods for both static and dynamic graphs. Specifically, we first develop a static MSCSC method that only needs one scan of the graph $G$, runs in linear time w.r.t., the number of edges, and provides rigorous approximation guarantees. Then, based on this, we leverage a reduced directed acyclic graph of $G$ to design incremental MSCSC method $\text{MSC}^3$ with two variants to handle edge insertions efficiently. We further develop $\text{MSC}^4$ that updates MSCSC under edge deletions by efficiently scanning only locally affected subgraphs. Moreover, to demonstrate the high utility, we conduct two use case studies to apply our MSCSC methods to boost the efficiency of dynamic strongly connected component (SCC) maintenance and dynamic SCC-based reachability index maintenance. Extensive experiments on 8 large graphs, including 3 billion-edge graphs, validate the superior efficiency of our methods.

PVLDB Reference Format:

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/jerchenxin/mscsc.

1 INTRODUCTION

In a directed graph $G$, nodes $u$ and $v$ are strongly connected if they are reachable from each other. Real-world graphs are often dynamically changing. Identifying and maintaining the strong connectivity information whenever graph $G$ changes with new edge insertion or deletion is a challenging but important task, which is useful in telecommunication networks [4], social community analysis [17, 30, 35, 43], and the design of dynamic indexes for important graph algorithms, e.g., dynamic reachability queries [50, 53, 56].

Given a directed graph $G$, a conventional way is to detect all the strongly connected components (SCCs), each of which is a maximal subgraph containing a set of nodes that are strongly connected to each other and all the edges among the nodes. For an SCC, no additional nodes from $G$ can be included in it without breaking its strong connectivity. Though linear-time SCC detection algorithms exist on static graphs [14, 42, 45], the dynamic maintenance of SCCs is expensive for two reasons. First, an SCC may contain redundant edges for strong connectivity, and updates on these redundant edges require costly dynamic maintenance but actually do not affect the strong connectivity between the nodes in the SCC. Further, it is shown [2] that the problem of deciding whether there are more than two SCCs in a fully dynamic graph cannot be solved with $O(m^{1+\epsilon})$ amortized time on sparse graphs for any $\epsilon > 0$, where $m$ is the number of edges, which theoretically indicates the expensive overheads of dynamic SCC maintenance.

To address the aforementioned issues, instead of maintaining SCC subgraphs, we propose a new problem, minimum strongly connected subgraph collection (MSCSC), which extends and enhances the problem of minimum strongly connected subgraph (MSCS) [48, 55]. Briefly, given a strongly connected graph, MSCS finds a spanning subgraph that contains all nodes of the graph and is strongly connected with the fewest edges. However, a real graph $G$ may not be strongly connected, and contains multiple MSCSs. Hence, in $G$, MSCSC finds a collection of all MSCSs, each of which contains a maximal set of nodes that are strongly connected via minimum number of edges in $G$. For example, for the graph in Fig. 1, the MSCSC is shown in red edges and it consists of two MSCSs. One MSCS is formed by the red edges connecting $v_1, v_2, v_3, v_4, v_5, v_6$, and the other is formed by the red edges connecting $v_7, v_8, v_9, v_{10}, v_{11}, v_{12}$. The black edges are not in the MSCSC.

Applications. One important utility of MSCSC is to speed up fundamental graph processing tasks. As mentioned, given a graph $G$, its SCCs may contain redundant edges for strong connectivity,
The mapping function between $A$ reduced graph $A$ directed graph $A$ directed edge from $An$ approximate MSCSC of $v$ while MSCSC only maintains the fewest edges to preserve the same strong connectivity. We have provided two use cases in Section 5.3 that leverages MSCSC to boost the efficiency of dynamic SCC maintenance and dynamic SCC-based reachability index maintenance, revealing the motivation of the study on MSCSC. Moreover, MSCSC is expected to be useful in telecommunication network monitoring and community analysis. For example, when Fig. 1 is a telecommunication network with its MSCSC in red, since nodes $v_1$ and $v_7$ are not connected by any edge in the MSCSC, the connection between $v_1$ and $v_7$ should be categorized as vulnerable to network interruptions. Further, if a sudden network interruption (edge deletion) happens on the red edge from $v_5$ to $v_6$, it will cause disconnectivity on the left MSCS, and it should be classified as a critical interruption to be fixed immediately. Contrarily, if a network interruption happens on the black edge from $v_2$ to $v_3$, it does not change the MSCSC (i.e., strong connectivity unchanged) and can be regarded as a non-critical issue to save maintenance cost.

**Challenges.** MSCSC computation is challenging, especially on massive dynamic graphs with millions of nodes and billions of edges. We show that MSCSC is NP-hard on static graphs. A trivial solution of MSCSC is to firstly detect the SCCs in graph $G$ by SCC methods [14, 42, 45], then apply existing MSCS methods [48, 55] on every SCC to detect MSCS, and finally union all edges in the detected MSCSs as $E_{nec}$. This solution requires scanning the input graph at least twice, and in experiments it is inefficient to maintain $E_{nec}$ when graph $G$ is dynamically changing. In literature, there exist studies for SCC maintenance [5–7, 28, 37, 53]. As for MSCS, existing studies mainly focus on static graphs to develop approximate solutions with strong theoretical guarantees [48, 55], while no dynamic MSCS methods exist. It is costly to re-identify MSCSs on all SCCs from scratch whenever graphs change. To the best of our knowledge, there exist no studies on dynamic MSCSC maintenance, and existing SCC and MSCS studies are inefficient to identify and maintain the MSCSC of dynamic graphs.

**Contributions.** To address the challenges, we define $\alpha$-approximate MSCSC to find an edge set $E_{nec}$ with size bounded by an approximation factor $\alpha$ on the size of the optimal solution in Section 2. Then, we develop a new 2-approximate MSCSC method $MSC^d$ that needs only one scan of the input graph and provides rigorous approximation guarantees (Section 3). Further, we design dynamic MSCSC maintenance methods to handle edge insertions and deletions in Section 4. Specifically, in Section 4.1, we leverage a reduced directed acyclic graph (DAG) of the input graph to design an incremental MSCSC method $MSC^c$ that only works on the locally affected subgraphs for MSCSC updates under new edge insertions. In particular, we develop two variants of $MSC^c$ with 2-approximation, one of which is optimal in terms of the number of edges added into approximate MSCSC $E_{nec}$ and the other is practically efficient. To handle edge deletions, in Section 4.2, we design $MSC^d$ that updates MSCSC by efficiently scanning only local subgraphs. All the methods run in linear time w.r.t. graph size. Extensive experiments (Section 5) on eight large graphs and two use cases demonstrate the superior efficiency and approximation effectiveness of our methods.

To sum up, we make the following contributions in our paper.

- We introduce the problem of MSCSC maintenance on dynamic graphs, which is useful in real applications. Given a directed graph, MSCSC aims to find a collection of maximal subgraphs each of which is strongly connected via the fewest edges.
- We develop an approximate solution $MSC^c$ that runs one scan of the graph to identify approximate MSCSC with rigorous guarantees.
- We then present $MSC^d$ and $MSC^c$ that are efficient in maintaining approximate MSCSC on dynamic graphs with edge updates, including insertions and deletions.
- We apply our methods to two use cases, dynamic SCC maintenance and dynamic reachability index maintenance, and conduct extensive experiments to validate the superiority of our methods.

### 2 PRELIMINARIES

#### 2.1 Problem Formulation

Let $G = (V, E)$ be a directed graph, where $V$ is the set of nodes with cardinality $n = |V|$, and $E$ is the set of edges with cardinality $m = |E|$. Nodes $u$ and $v$ are strongly connected if there exist a path from $u$ to $v$ and a path from $v$ to $u$ in $G$.

A strongly connected component (SCC) of $G$ is defined as a maximal subgraph of $G$ where any two nodes are reachable to each other in the subgraph. There can be multiple SCCs in a graph $G$. Supposing that $G$ is a strongly connected graph (i.e., $G$ itself is an SCC), the problem of minimum strongly connected spanning subgraph (MSCS) is to find a strongly connected subgraph containing all nodes in $G$ but with the fewest edges. A real graph $G$ may contain multiple SCCs. We extend MSCS to such real graphs, and propose to study a new problem, *minimum strongly connected subgraph collection (MSCSC)* defined below.

**Definition 2.1 (MSCSC).** Given an input graph $G$, MSCSC aims to find a collection of MSCSs, each of which is a subgraph that contains a maximal set of nodes that are strongly connected from each other via the fewest edges. Let $E_{nec}$ be the set of edges in the optimal MSCSC solution for $G$.  

<table>
<thead>
<tr>
<th>Table 1: Frequently used notations.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Notation</strong></td>
</tr>
<tr>
<td>$G = (V, E)$</td>
</tr>
<tr>
<td>$n$, $m$</td>
</tr>
<tr>
<td>$G' = (V', E')$</td>
</tr>
<tr>
<td>$f(\cdot)$</td>
</tr>
<tr>
<td>$E_{nec}, E_{nec}^{opt}$</td>
</tr>
<tr>
<td>$\alpha$</td>
</tr>
<tr>
<td>$GC, S$</td>
</tr>
<tr>
<td>$(u, v)$</td>
</tr>
<tr>
<td>$(u', v')$</td>
</tr>
</tbody>
</table>

![Figure 1: Red edges form the MSCSC containing two MSCSs](image-url)
we define dynamic MSCSC maintenance as follows. Tarjan’s SCC algorithm [45].

\[
\text{if } \text{visited}[u] = \text{false} \forall v \in V
\]

for each vertex \( u \in V \) do

\[
\text{if visited}[u] = \text{false then}
\]

\[
\text{DFS}(u)
\]

return SCCs

Procedure DFS

\[
\text{low}(u) \leftarrow \text{depth}, \text{dfn}(u) \leftarrow \text{depth}
\]

\[
\text{S.push}(u), \text{visited}[u] \leftarrow \text{true}, \text{depth} \leftarrow \text{depth} + 1
\]

for each outgoing edge \((u,v)\) of \( u \) do

\[
\text{if visited}[v] = \text{false then} \quad \text{// case 1}
\]

\[
\text{DFS}(v)
\]

\[
\text{low}(u) \leftarrow \min\{\text{low}(u), \text{low}(v)\}
\]

else if \( v \in S \) then

\[
\text{low}(u) \leftarrow \min\{\text{low}(u), \text{dfn}(v)\}
\]

\[
\text{if } \text{low}(u) = \text{dfn}(u) \text{ then} \quad \text{// create an SCC}
\]

Pop all elements in stack \( S \) until it reaches \( u \) and add them to an SCC \( S \). Add \( S \) to SCCs.

Build the node-to-SCC mapping function \( f(w) = S \) from every node \( w \) to \( S \).

The MSCSC in Fig. 1 is formed by the red edges. \( E_{\text{nec}}^{\text{opt}} \) is the set of red edges. Intuitively, all edges in \( E_{\text{nec}}^{\text{opt}} \) are necessary to keep the strong connectivity of all MSCSs in \( G \). If two nodes are strongly connected in \( G \), they are still strongly connected via edges in MSCSC \( E_{\text{nec}}^{\text{opt}} \). Naturally, these edges in MSCSC are called necessary edges. Deleting a necessary edge may disconnect certain nodes in MSCSC, while deleting any edge outside \( E_{\text{nec}}^{\text{opt}} \) will not affect the strong connectivity information of the input graph \( G \).

MSCS itself is NP-hard [55]. For each SCC in \( G \), MSCSC will find an MSCS. Thus, it is NP-hard to find an optimal solution \( E_{\text{nec}}^{\text{opt}} \) of MSCSC in \( G \). Hence, we focus on \( \alpha \)-approximate MSCSC.

Definition 2.2 (\( \alpha \)-Approximate MSCSC). Given an input graph \( G \), an approximate MSCSC solution \( E_{\text{nec}} \) is a necessary edge set of size bounded by an approximation factor \( \alpha \) over the size of the optimal \( E_{\text{nec}}^{\text{opt}} \), i.e., \( |E_{\text{nec}}|/|E_{\text{nec}}^{\text{opt}}| \leq \alpha \).

On dynamic graphs that may have edge insertions and deletions, we define dynamic MSCSC maintenance as follows.

Definition 2.3 (Dynamic MSCSC Maintenance). Given an input graph \( G \) with an approximate MSCSC solution \( E_{\text{nec}} \), dynamic MSCSC maintains the up-to-date \( E_{\text{nec}} \) when edges are inserted or deleted.

Tab. 1 displays the frequently used notations in this paper.

2.2 Existing Solutions on SCC and MSCS

In this section, we review existing SCC and MSCS methods.

Tarjan’s SCC algorithm [45]. There exist algorithms to efficiently find SCCs [14, 42, 45]. Tarjan’s algorithm [45] is one representative method, with its pseudo-code in Algo. 1. The whole algorithm runs in a depth-first search (DFS) manner. Initially, at Line 1, it sets \( depth \) to be 1, which is a global counter to be incremented by one when a new node is visited, and maintains a flag \( visited \) per node, recording whether the node has been visited or not and initialized to be false. A global stack \( S \) is used to detect SCCs (Line 2). For every unvisited node \( u \), it triggers a DFS procedure to identify SCCs (Lines 3-5). In the DFS procedure (Line 8 in Algo. 1), every node \( u \) maintains a value \( dfn(u) \), which records the visiting order of \( u \) in the DFS traversal. If \( u \) is the \( i \)-th visited node during the DFS, then \( dfn(u) = i \). Node \( u \) further has a value \( low(u) \) to record the current smallest \( dfn(v) \) value among all nodes that are reachable from \( u \). At Line 9, initially, both \( low(u) \) and \( dfn(u) \) are set to \( depth \), and \( u \) is pushed into the stack \( S \). The main idea is that if a node \( v \) is the first node visited among all nodes in an SCC, then node \( v \) must have the smallest \( dfn(v) \) value among all nodes in the SCC, and \( dfn(v) \) also equals to \( low(v) \). The global stack \( S \) is used to find all nodes in the same SCC. In Tarjan’s algorithm, for the nodes in the same SCC as \( v \), they will be on top of \( v \) in the stack \( S \). Then, we can retrieve the SCC containing \( v \) via popping all elements in \( S \) until \( v \) is popped out. In particular, after marking \( u \) as visited and increasing \( depth \) by 1 at Line 10, for every out-neighbor \( v \) of \( u \) (Line 11), if \( v \) is not visited yet, recursive DFS is applied (Line 13), after which, the \( low(v) \) value is updated if \( low(v) < low(u) \) (Line 14). Otherwise, \( v \) has already visited, and if \( v \) is already in \( S \) (Line 15), the \( low(u) \) value is also updated if the \( dfn(v) \) value is smaller. After performing DFS of all out-neighbors of \( u \) from Lines 11 to 16, at Line 17, if \( u \) is the first node visited in an SCC (i.e., \( low(u) = dfn(u) \)), then a new SCC \( S \) is discovered and all nodes above \( u \) (including \( u \) in \( S \)) are popped out and added to \( S \) (Lines 18-19). The time and space complexities of Algo. 1 are both \( O(n + m) \).

Dynamic SCCs. In literature, there exist studies for SCC maintenance [5-7, 28, 37, 35]. As mentioned, the problem of whether there are more than two SCCs in a fully dynamic graph cannot be solved with \( O(m^{1-\epsilon}) \) amortized update and query times on sparse graphs for any \( \epsilon > 0 \) [2]. Thus, existing dynamic SCC studies mainly focus on the partially dynamic setting: either the decremental setting, where there are only edge deletions [7, 28], or the incremental setting, where there are only edge insertions [5, 6].

MSCS. There exist several studies to find approximate MSCS [25, 26, 48, 55]. A super-linear-time 1.64-approximation algorithm is presented in [25], and Khuller et al. [26] develop a super-linear-time algorithm with an approximate ratio of about 1.61. Vetta et al. [48] present a super-linear-time 3/2-approximation algorithm. Note that all these three methods run in super-linear time that is higher than linear time. Zhao [55] is a linear-time 5/3-approximation algorithm, with pseudo-code shown in Algo. 2. This algorithm repeatedly
We first provide the definitions and conduct approximation analysis when performing DFS traversal from a visited node. Let \( \delta^* \) of a node \( u \) correspond to \( \delta^*(u) \), where \( \delta^*(u) \) is the set of outgoing edges of nodes in \( u \). Fig. 2 presents a running example of Zhao's method. Graph \( G \) contains two SCCs: \( (v_0) \) and \( (v_1, v_2, v_3, v_4, v_5) \). On the large SCC, to detect MSCSC, Zhao first finds a concealing cycle formed by the red edges in Fig. 2(a), and each edge of this cycle is marked as a necessary edge. Then Zhao contracts the cycle as a node \( v' \) and forms a graph \( G_z \) in Fig. 2(b), where \( \langle v_1, v'_2 \rangle \) and \( \langle v_3, v_1 \rangle \) correspond to \( \langle v_2, v_3 \rangle \) and \( \langle v_3, v_4 \rangle \) in \( G \), respectively. As there is no concealing cycle of at least 3 in \( G_z \), Hao marks edges inside all 2-cycles in \( G_z \) as necessary. In Fig. 2(b), there is a 2-cycle formed by \( v_1 \) and \( v'_2 \), and red edges inside this cycle become necessary. Then, Hao finds the necessary edges for the SCC, including \( \langle v_1, v'_2 \rangle \), \( \langle v_3, v_4 \rangle \), \( \langle v_4, v_3 \rangle \), \( \langle v_5, v_3 \rangle \).

Existing MSCS methods, e.g., [25, 55], can be extended to handle MSCSC by first detecting all SCCs in \( G \) and then finding the MSCSC of each SCC, which requires scanning \( G \) at least twice and is inefficient. Moreover, existing MSC solutions are designed for static graphs, and inefficient in maintaining dynamic MSCSC. Thus, there is an urgent need for efficient dynamic MSCSC maintenance.

3 APPROXIMATE MSCSC

We first provide the definitions and conduct approximation analysis in Section 3.1 to present our 2-approximation guarantee on MSCSC. Then we develop the algorithmic details of the 2-approximate method MSC in Section 3.2. MSC only needs one scan of \( G \) to identify an approximate necessary edge set \( E_{\text{necc}} \) of MSCSC. MSC is the basis of the dynamic methods developed later in Section 4.

3.1 Definitions and Approximation Analysis

To facilitate the designs in our method, we define two types of edges, namely tree edges and dropping edges, which are essential to get an approximate MSCSC \( E_{\text{necc}} \) of a graph \( G \).

For the ease of understanding, in the following, we focus on the analysis on a strongly connected graph \( G \). The approximation analysis is extended to graph \( G \) that may not be strongly connected in Theorem 3.3. Definition 3.1 defines tree edges, which are the edges in the DFS tree generated in the depth-first traversal process.

**Definition 3.1 (Tree edge).** Given a strongly connected graph \( G \), when performing DFS traversal from a visited node \( u \), an edge \( (u, v) \) is a tree edge if \( u \) reaches an unvisited node \( v \) via edge \( (u, v) \) and \( u \) and \( v \) are strongly connected.

Further, as shown in Algo. 1, an edge \( (u, v) \) can cause the drop of \( low(u) \) value, if \( low(v) \) or \( dfn(v) \) is smaller (Lines 14 or 16), which indicates that \( u \) can reach certain nodes \( v \) that have already been visited and they belong to the same SCC. Therefore, for each node \( u \), we track those out-going edges \( (u, v) \) that alter the value of \( low(u) \). We denote such edges as dropping edges, defined as follows.

**Definition 3.2 (Dropping edge).** Given a strongly connected graph \( G \), we denote the edge \( (u, v) \) that causes the drop of \( low(u) \) of node \( u \) as a dropping edge of \( u \).

Let \( E_{\text{tree}}(G) \) and \( E_{\text{drop}}(G) \) be the sets of tree edges and dropping edges in \( G \), respectively. In Lemma 1, we prove that, the union of all tree edges and dropping edges in \( G \) preserves the strong connectivity of any two nodes in \( G \). All proofs can be found in the full-version technical report [1].

**Lemma 1.** For a strongly connected graph \( G \), \( E_{\text{tree}}(G) \cup E_{\text{drop}}(G) \) preserves the strong connectivity between any nodes in \( G \).

However, note that \( E_{\text{tree}}(G) \cup E_{\text{drop}}(G) \) does not have an approximation guarantee with respect to the optimal solution \( G \), since a node \( u \) can have multiple dropping edges. In other words, the value \( low(u) \) may be changed more than once. For example, in Fig. 3, the value \( low(u) \) is changed from 4 to 6 (due to edge \( (u_4, u_5) \) and then to 2 (due to edge \( (u_4, u_2) \)). In the worst case, a node \( u \) may have its \( low(u) \) value changed once for every out-going edge (i.e., all its out-going edges are dropping edges).

To address the issue, we propose to only consider the last dropping edge of a node, which can reduce the number of necessary edges significantly. Given a node \( u \) with multiple dropping edges, we only keep the last edge that changes the value of \( low(u) \). Hence we maintain a last dropping edge set \( E_{\text{lastdrop}}(G) \), without losing the strong connectivity information as proved in Lemma 2. Note that tree edges are necessary to keep the full connectivity information, and we will keep the tree edges as discarding any of them might cause the loss of connectivity information. Also, if there is a tree edge \( (u, v) \) which produces the same \( low(u) \) value as the last dropping edge of \( u \), we can discard such a last dropping edge without affecting the strong connectivity, and further reduce the number of necessary edges maintained.

In Lemma 2, we first prove that every node \( u \) can reach the first node \( r \) starting the DFS in \( G \) via last dropping edges only \( E_{\text{lastdrop}}(G) \). Then it is natural to derive Lemma 3 that \( E_{\text{tree}}(G) \) and \( E_{\text{lastdrop}}(G) \) together can preserve the strong connectivity of any two nodes in \( G \).

**Lemma 2.** Given a strongly connected graph \( G \), every node \( u \) in \( G \) can reach node \( r \) that is the first node visited during DFS in \( G \), via the last dropping edges in \( E_{\text{lastdrop}}(G) \).

**Lemma 3.** \( E_{\text{tree}}(G) \cup E_{\text{lastdrop}}(G) \) preserves the strong connectivity between any nodes in a strongly connected graph \( G \).

Finally, for any graph \( G \) that may not be strongly connected, in Theorem 3.3, we derive that the last dropping edges and tree edges of a graph \( G \) together form a 2-approximation MSCSC solution \( E_{\text{necc}} \) w.r.t., \( E_{\text{opt}} \).

**Theorem 3.3.** Given a graph \( G \), for every SCC with its tree edges and last dropping edges in \( G \), let necessary edge set \( E_{\text{necc}} \) be the union of all tree edges and last dropping edges in \( G \). \( E_{\text{necc}} \) is a 2-approximation MSCSC w.r.t., the optimal, i.e., \( |E_{\text{necc}}|/|E_{\text{opt}}| \leq 2 \).
Algorithm 3: Approximate MSCSC: MSC

Input: Directed graph G
Output: Approximate MSCSC \( E_{\text{approx}} \)

1. \( E_{\text{approx}} \leftarrow \emptyset \)
2. \( \text{lastdrop} \leftarrow \emptyset \)
3. \( \text{depth} \leftarrow 1 \), \( \text{visited}(v) \leftarrow \text{false} \) \( \forall \ v \in V \)
4. for each node \( u \in V \) do
5.     if \( \text{visited}(u) = \text{false} \) then
6.         \( \text{ProcessNode}(u) \)
7. \( E_{\text{approx}} \leftarrow \text{lastdrop} \cup \left\{ (u,v) \in E_{\text{approx}} \mid f(u) = f(v) \right\} \)
8. end for

3.2 Algorithm

Algo. 3 and 4 present the pseudo code of our 2-approximation MSCSC solution MSC on static graphs. Remark that MSC also adopts DFS traversal with similar symbols in Algo. 1, but with vital new designs to efficiently achieve the approximation guarantees in Theorem 3.3 based on the newly proposed tree edges and dropping edges in Definitions 3.1 and 3.2, compared with Algo. 1.

After initialization (Lines 1-2 of Algo. 3), for every unvisited node \( u \), we perform procedure ProcessNode (Algo. 4). In Algo. 4, Lines 1-10 are with the same initialization as Algo. 1. At Line 11, \( \text{lastdrop} \) represents the last dropping edge of node \( u \) and is set to empty initially. When node \( u \) reaches an unvisited node \( v \) (Lines 5-10 of Algo. 4), this edge will be temporarily marked as a tree edge (note that edges that are not in the same MSC will be excluded from \( E_{\text{approx}} \) in the end at Line 6 of Algo. 3). After executing the procedure recursively for \( v \) at Line 7, if \( \text{low}(u) \geq \text{low}(v) \) (Line 8), indicating that we can produce the low value which is at least no greater than the previous one, it then updates this tree edge as the last dropping edge of node \( u \) (Line 9). In Lines 11-13 of Algo. 4, when node \( u \) reaches a visited node \( v \) which is still in the stack and \( \text{low}(u) > \text{dfn}(v) \), this edge is updated as the last dropping edge \( \text{lastdrop} \) of \( u \). At the end of Algo. 4, it adds the last dropping edge into \( E_{\text{lastdrop}} \) (Lines 14-15 of Algo. 4) and generates a new MSC (Line 16 of Algo. 4). At the end of Algo. 3 (Line 6), it collects all necessary edges by first excluding false tree edges (\( u,v \) not in the same MSC, i.e., \( f(u) \neq f(v) \)) from \( E_{\text{approx}} \), and then taking the union of \( E_{\text{approx}} \) and \( E_{\text{lastdrop}} \) (Line 6 of Algo. 3).

The time and space complexities of Algo. 3 are both \( O(n+m) \), as the graph traversal (procedure ProcessNode) visits each node and edge exactly once, with a consistent time-space cost per edge.

Example 3.4. Fig. 3 shows an example to get MSCSC by MSC (red edges). Fig. 3(a) is the result after processing \((v_1, v_2), (v_2, v_3)\) and \((v_3, v_1)\). \((v_1, v_2)\) and \((v_2, v_3)\) are added to \( E_{\text{approx}} \) as tree edges, while \((v_3, v_1)\) is a dropping edge since \( \text{low}(3) \) is changed from 3 to 1. It is now marked as the last dropping edge of \( v_3 \). In Fig. 3(b), we visit \((v_3, v_4)\) and \((v_4, v_5)\). Edge \((v_3, v_4)\) is added to \( E_{\text{approx}} \) as it is a tree edge. When we reach \((v_4, v_5)\), \( \text{low}(v_5) \) is dropped and thus it is a dropping edge of \( v_4 \). We set \((v_4, v_5)\) temporarily as the last dropping edge of \( v_4 \). In Fig. 3(c), we deal with \((v_4, v_2)\). We will prune the previously stored last dropping edge \((v_4, v_5)\) since \( \text{low}(v_5) \) is now updated again and edge \((v_4, v_2)\) becomes the new last dropping edge. After processing \((v_4, v_5)\) and \((v_5, v_2)\) in Fig. 3(d), \((v_5, v_2)\) is added to \( E_{\text{approx}} \) and \((v_5, v_2)\) is set as the last dropping edge of \( v_5 \) since \( \text{low}(v_5) \) is changed. Also, the last dropping edge of \( v_4 \) is updated from \((v_4, v_2)\) to \((v_4, v_5)\) since \( \text{low}(v_4) = \text{low}(v_5) \) and \((v_4, v_5)\) is a tree edge. Finally, a new MSC is formed by \( v_1, v_2, v_3, v_4, \) and \( v_5 \). Also, \( v_6 \) forms another MSC. And, we have \( E_{\text{approx}} = \{(v_1, v_2), (v_2, v_3), (v_1, v_4), (v_4, v_5)\} \). The set \( E_{\text{lastdrop}} \) of last dropping edges is \{\((v_3, v_1), (v_2, v_3), (v_3, v_5), (v_2, v_5)\)\}. Unioning \( E_{\text{approx}} \) and \( E_{\text{lastdrop}} \), we get \( E_{\text{approx}} \) (red edges in Fig. 3(d)).

4 DYNAMIC MSCSC MAINTENANCE

In Section 4.1, we present two incremental maintenance methods for edge insertions a 2-approximation, one is optimal in terms of the number of edges added into approximate MSCSC \( E_{\text{approx}} \) and the other is practically efficient. Then in Section 4.2, we develop a decremental maintenance method to handle edge deletions.

4.1 Incremental Update

When edges are inserted into \( G \), a way is to directly work on graph \( G \) with the new edge to detect the new MSCSs, which is inefficient. Instead, we leverage a reduced directed acyclic graph (DAG) \( G' \) for efficient MSCSC maintenance. Given the input graph \( G \), after getting approximate MSCSC \( E_{\text{approx}} \) (i.e., detecting all MSCSs), we build a DAG \( G' \), where all nodes \( v_i \) in an MSCS of \( G \) are mapped to a single node \( f(v_i) \) in \( G' \), where \( f \) is the mapping function between \( G \) and \( G' \) (i.e., an MSCS in \( G \) is a node in \( G' \)). We use \( v' \) and \( E' \) to represent the node set and edge set of \( G' \) respectively. There is an edge from \( u' \) to \( v' \) in \( G' \), if there is at least one edge from any node in the MSCSs of \( u' \) to any node in the MSCSs of \( v' \) in \( G \).

Our incremental methods first work on \( G' \) and then map back on \( G \) to maintain the approximate MSCSC \( E_{\text{approx}} \). Given a new edge \((u_i, v_i)\) inserted into \( G \), if \( u_i \) and \( v_i \) belong to the same MSCS (i.e., \( f(u_i) = f(v_i) \)), then the approximate MSCSC \( E_{\text{approx}} \) does not change, since \( u_i \) and \( v_i \) are already strongly connected via \( E_{\text{approx}} \). If \( u_i \) and \( v_i \) belong to different MSCSs (i.e., \( f(u_i) \neq f(v_i) \)), then the insertion of the new edge may cause the merge of MSCSs. Fig. 4 shows an example of DAG \( G' \) obtained by reducing the MSCSC.
of a graph $G$. If a new edge $(u_i, v_i)$ is inserted into $G$ and $f(u_i) = v'_i, f(v_i) = v''_i$ in $G'$ (i.e., $u_i$ and $v_i$ are in different MSCs), then it means that a corresponding edge in blue $(v'_i, v''_i)$ is also inserted into $G'$. Observe that the edge may cause the merge of the MSCs. We first identify the strongly connected nodes in $G'$, and consequently obtain the MSCs that should be merged in $G$, and finally update $E_{necc}$ accordingly in $G$, after a new DAG $G'$ is also obtained.

**Optimal 2-Approx Incremental MSCSC $MSC^{1\star}$.** $MSC^{1\star}$ is optimal in the sense that, given a new edge insertion, the number of edges added in $E_{necc}$ for the insertion is minimum. In other words, removing any one of these newly added edges will cause disconnectivity of nodes in MSCSC. We first identify an SCC $S'$ in the new DAG $G'$. $S'$ must contain the new edge $(u', v')$, where $f(u_i) = u'$, $f(v_i) = v'$ and $(u_i, v_i)$ is the new edge in $G$. Then denote $G'' = S' \setminus (u', v')$. Apparently $G''$ is a DAG. In fact, for Fig. 4, DAG $G''$ and the new edge in blue form an SCC. In this example, $G''$ is itself without the new edge. In DAG $G''$, only node $v'$ (resp. $u'$) is with zero in-degree (resp. out-degree), e.g., $v'_i$ and $u'_i$ respectively in Fig. 4. Further, all other nodes in $G''$ are on the paths from $v'$ to $u'$. In the optimal solution, we conduct traversal from $v'$ via all paths to $u'$, and develop a topological sort technique to only mark the edges that are essential to maintain the connectivity of all nodes to $u'$. In the traversal over $G''$, for every node $v'_j$ (except $v'$ and $u'$), it will have only one incoming edge as well as only one outgoing edge marked as necessary, which in the end will be combined as the optimal necessary edge set $E_{necc}'$ of $G''$. For every edge $e' \in E_{necc}'$ on the reduced graph, there can be many edges in the original graph $G$ corresponding to it, among which, we simply choose one edge $e$ arbitrarily and add it into $E_{necc}$.

Algo. 5 presents the pseudo code of $MSC^{1\star}$. The input includes the reduced DAG $G''$, and a new edge $(u', v')$ (corresponding to a new edge $(u_i, v_i)$ in graph $G$). The output is the updated $E_{necc}$ and DAG $G''$. Algo. 5 first detects if there is a new SCC $S'$ in the new $G''$ (Line 2). If no new SCC, then nothing to be performed (Lines 3-4). Otherwise, we aim to identify the MSCS $E_{necc}'$ of SCC $S'$. Specifically, we first get $G'$ at Line 5. $G'$ is a DAG with paths from $v'$ to $u'$, but not the other way around. Then for every node $v'_j$ in $G''$, we initialize a flag, reach, to indicate whether it is reachable from $v'$ (Line 6). We then get the in-degree of every node $v'_j$ in $G''$ (Line 7). We maintain a queue $Q$ to start the traversal from $v'$ (Line 8). For every node $v'_j$ popped from $Q$ (Lines 10-11), we first maintain a fake reach$U_i$ to indicate if it has any out-going edge added into $E_{necc}'$ (i.e., marked as necessary), which is initialized as false (Line 11). Then at

![Figure 3: Running example for our method MSC.](https://example.com/fig3)

**Algorithm 5: Optimal Incremental MSCSC: $MSC^{1\star}$**

**Input:** Graph $G$ with approximate MSCSC $E_{necc}$, and the corresponding DAG $G''$, a new edge inserted into $G$ that maps to a new edge $(u', v')$ in $G''$.

**Output:** Updated $E_{necc}$ and a new $G''$

1. Add $\langle u', v' \rangle$ into $G''$
2. Invoke the procedure $DFS'(v')$ in Algo. 1 from root $v'$ in $G''$, to detect if a new SCC is formed due to edge $(u', v')$
3. if no new SCC then
   4. return
5. $G' \leftarrow S' \setminus \langle u', v' \rangle$, $E_{necc} \leftarrow \{\langle u', v' \rangle\}$
6. reach$[v'_i] = false \forall nodes v'_i \in G''$
7. Get $d_{in}[v'_i]$ for each $v'_i$ in $G''$
8. Queue $Q.push(v')$
9. while $Q$ is not empty do
10. pop $v'_i$ from $Q$
11. reach$U_i \leftarrow false$
12. for each outgoing edge $\langle v'_j, v'_k \rangle$ of $v'_i$ in $G''$ do
13. $d_{in}[v'_j] \leftarrow d_{in}[v'_j] - 1$
14. if $d_{in}[v'_j] = 0$ then
15. $Q.push(v'_j)$
16. if reach$[v'_j] = false$ then
17. reach$U_i \leftarrow true$
18. reach$[v'_j] \leftarrow true$, $E_{necc}.add(\langle v'_j, v'_k \rangle)$
19. if reach$U_i = false$ then
20. let $\langle v'_j, v'_k \rangle$ be one of outgoing edges of $v'_j$
21. reach$[v'_j] \leftarrow true$, $E_{necc}.add(\langle v'_j, v'_k \rangle)$
22. Produce a new $G''$ by shrinking $S'$ into a node
23. for each edge $e' \in E_{necc}'$ do
24. Add one of edges $e$ in $G$ that maps to $e'$ into $E_{necc}$

Line 12, we iterate every out-going neighbor $v'_j$ of $v'_i$ (i.e., out-going edge) to check if the edge is the last-visited incoming edge of $v'_k$ by decreasing the in-degree count of $v'_k$ (Line 13). If the count becomes zero, it means all incoming edges of $v'_k$ have been traversed, and $(v'_j, v'_k)$ is the last one (Line 14). Consequently, we push $v'_k$ into the queue (Line 15). If $v'_k$ is not determined as reachable from $v'$ (Line 16), we add the edge into $E_{necc}'$ and mark both reach$[v'_j]$ and reach$U_i$ as true. After inspecting all outgoing edges of $v'_j$ (Lines 12-18), if reach$U_i$ is still false (Line 19), it means that none of $v'_j$'s
outgoing edges is added into $E'_{necc}$, then we just pick one outgoing edge and added it to $E'_{necc}$ as well as marking the corresponding next-neighbors as reached at Lines 20-21. Then the nodes in $S'$ of $G'$ with the new edge shrink to a node such that we can get a new DAG $G'$ (Line 22). For every edge $e'$ in $E'_{necc}$, we choose one edge $e$ in $G$ that maps to $e'$ and insert $e$ into $E_{necc}$, which is the updated MSCSC for inserting new edge $\langle u'_{i}, v'_{i} \rangle$ into $G$ (Lines 23-24).

Example 4.1. Suppose an edge $\langle u_{i}, v_{i} \rangle$ is added to the input graph where $f(u_{i}) = u'_{i}$ and $f(v_{i}) = v'_{i}$ in Fig. 4. Then, we add edge $\langle u'_{i}, v'_{i} \rangle$ (shown in blue) to the reduced graph $G'$. In $G'$, we first run Algo. 1 to find the new SCC, which consists of all nodes in Fig. 4 (i.e., $G'$ is the DAG without the blue edge). Then we start the topological sort from $u'_{i}$ on $G'$, since $u'_{i}$ is the only node with $d_{in} = 0$. Initially, only $u'_{i}$ is in $Q$. Then, we pop $u'_{i}$ from $Q$ and update $d_{in}$ of $v'_{i}$ and $v'_{j}$. We find that $d_{in}[v'_{i}]$ (resp. $d_{in}[v'_{j}]$) becomes zero. As $v'_{i}$ (resp. $v'_{j}$) has only one incoming edge and reach[$v'_{i}$] (resp. reach[$v'_{j}$]) is false, $\langle v'_{i}, v'_{j} \rangle$ becomes necessary and reach[$v'_{i}$] (resp. reach[$v'_{j}$]) becomes true. Also, we push $v'_{j}$ (resp. $v'_{i}$) into $Q$. Now, since reach[1] of $v'_{i}$ becomes true, we can start the next iteration. Next, we pop $v'_{j}$ from $Q$ and update $d_{in}$ of $v'_{i}$ and $v'_{j}$. Now $d_{in}[v'_{i}] = 0$. Since $\langle v'_{j}, v'_{i} \rangle$ is the only incoming edge of $v'_{i}$ and reach[$v'_{i}$] = false, this edge becomes necessary and reach[$v'_{i}$] becomes true. Also, we push $v'_{i}$ into $Q$. Besides, since reach[1] of $v'_{i}$ becomes true, we can turn to the next iteration. Next, we pop $v'_{j}$ from $Q$ and update $d_{in}$ of $v'_{j}$ which is not zero. We find that reach[1] of $v'_{j}$ is still false. Then we mark one arbitrary outgoing edge, say $\langle v'_{j}, v'_{k} \rangle$ as necessary and reach[$v'_{k}$] becomes true. Then, we pop $v'_{j}$ from $Q$ and update $d_{in}$ of $v'_{j}$. We find that $d_{in}[v'_{k}]$ becomes zero. Though $d_{in}[v'_{j}] = 0$, since reach[1] is true, we will not immediately mark this edge as necessary in Line 14. Yet, since we find that reach[1] of $v'_{i}$ is still false, we mark an outgoing edge $\langle v'_{i}, v'_{k} \rangle$ as necessary. Also, we push $v'_{i}$ into $Q$. At last, the topological sort ends with popping $v'_{j}$ from $Q$. Finally, we get $E'_{necc} = \{(v'_{i}, v'_{j}), (v'_{j}, v'_{i}), (v'_{i}, v'_{k}), (v'_{k}, v'_{j}), (v'_{j}, v'_{k})\}$. For each edge $e' \in E'_{necc}$, we choose an arbitrary edge in $G$ that maps to $e'$ and add it to the $E_{necc}$ of $G$.

Analysis of MSC$^4$. We prove that for DAG $G'$ with new edge $\langle u', v' \rangle$, if there is a new SCC $S'$ formed, the $E_{necc}$ identified by Algo. 5 is actually an optimal MSCS of $S'$, which is achieved by leveraging the DAG property of $G'$. As $E'_{necc}$ is an optimal MSCS in $G'$, it indicates that the number of edges added into the updated $E_{necc}$ is minimum. Then in Theorem 4.2, we prove the approximate guarantee in terms of incremental MSCSC maintenance. The time and space complexities are both $O(n' + m')$, where $n'$ and $m'$ are the number of nodes and edges in $G'$, as Algo. 5 firstly needs to run Algo. 1 to find the new SCC in $G'$ and then conduct the topological sort to locate necessary edges, which indicates that it needs to traverse $G'$ twice. Even so, this method is still more efficient than building from scratch, since this method only works in $G'$ whose size is much smaller than $G$. Besides, building from scratch with Algo. 5 can not provide an exact MSCSC solution, since Algo. 5 only guarantees that the number of edges added into $E_{necc}$ is minimum.

Lemma 4. In the DAG $G'$, suppose that there is a new SCC $S'$ after inserting an edge $\langle u', v' \rangle$, then the output edge set $E_{necc}$ identified by Algo. 5 is an optimal MSCS of $S'$.

Theorem 4.2. Given a graph $G$ with approximate MSCSC $E_{necc}$, and the corresponding DAG $G'$, a new edge inserted into $G$ that maps to a new edge $\langle u', v' \rangle$ in $G'$.

Algorithm 6: Incremental MSCSC: MSC$^4$

Input: Graph $G$ with approximate MSCSC $E_{necc}$, and the corresponding DAG $G'$, a new edge inserted into $G$ that maps to a new edge $\langle u', v' \rangle$ in $G'$

Output: Updated $E_{necc}$ and a new $G'$

```
1 af ← 0, E'_{necc} ← 0
2 if MergeMSCS(\langle u', v' \rangle) then
3     Merge vertices in af into a new MSCS
4     Produce a new $G'$ by shrinking $S'$ into a node
5     $E_{necc}$-add(\langle u', v' \rangle)
6     for each edge $e' \in E_{necc}$ do
7         Add one of edges $e$ in $G$ that maps to $e'$ into $E_{necc}$
8
9 Procedure MergeMSCS(\langle u', v' \rangle)
10    visited[u'] ← true
11    if u' = v' then // reach u'
12        return true
13    return false
14 $R ← false$
15    for each edge $\langle u'', v'' \rangle \in G'(\langle u', v' \rangle)$ do
16        if visited[u''] = true then // case 1
17            R ← true
18            if v'' \in af then
19                $af.add(u'')$
20                $E_{necc}$.add(\langle u'', v'' \rangle)
21            else if MergeMSCS(\langle u'', v'' \rangle) then // case 2
22                $R ← true$
23                $af.add(u'')$
24                $E_{necc}$.add(\langle u'', v'' \rangle)
24 return $R$
```

2-Approx Incremental MSCSC MSC$^4$. In the following, we present a more efficient 2-approximate solution MSC$^4$ in Algo. 6. MSC$^4$ does not require SCC detection. The method leverages the DAG properties of $G'$. The idea is that, any circle that makes any two nodes in $G' \cup \langle u', v' \rangle$ to be strongly connected must go through the new edge. Hence, if we find all paths from $u'$ to $u''$ in DAG $G'$, then we can locate all nodes in the paths in $G'$ to be merged. In this way, we do not need to maintain auxiliary information like live, dfn, and the stack $S$ in previous methods. Algorithms 6 provides the pseudo code of MSC$^4$, which performs in a DFS manner starting from $u'$. Specifically, at Line 1 in Algo. 6, $af$ is initialized to store
Algorithm 7: Decremental MSCSC: $MSC^{d}$

**Input:** $G$, deleted edge $(u_d, v_d)$, $E_{tree}$, $E_{lastdrop}$, $E_{nec}$

**Output:** Updated MSCSC $E_{nec}$

1. Delete this edge from $G$
2. if $(u_d, v_d) \notin E_{nec}$ then
3. \hspace{1cm} return
4. Get the all nodes in the same MSCS of $f(u_d)$, and retrieve the induced subgraph of the nodes, i.e., an SCC $G_S$, from $G$
5. $depth \leftarrow 1$, $visited[v] = false \forall v \in V(G_S)$, $redo \leftarrow false$
6. if SplitMSCS($u_d$) = false then
7. \hspace{1cm} Go to Line 11
8. for each vertex $v \in G_S$ do
9. \hspace{1cm} if visited[$u$] = false then
10. \hspace{2cm} ProcessNode($u$)
11. \hspace{1cm} $E_{nec} \leftarrow E_{lastdrop} \cup \bigcup_{(u, v) \in E_{tree}(f(u)) \cup f'(v)} (u, v)$

the nodes in $G'$ to be merged (the nodes correspond to the MSCSs to be merged in $G$), and $E'_{nec}$ contains the identified necessary edges in $G'$, which are going to be mapped back to the edges in $G$ to update $E_{nec}$. Procedure MergeMSCS is called at Line 2 in Algo. 6 to obtain $aff$ and $E'_{nec}$ for updates at Lines 3-7. Specifically, at Line 10, initially, node $v'_f$ is marked as visited. If the current node $v'_f$ is $u'$, which is the starting node of the new edge, then we add $v'_f$ into $aff$ and return true as the termination condition of recursion. A flag $R$ indicating to merge or not is initialized as false at Line 14. For every outgoing edge $(v'_f, v'_o)$ of $v'_f$ in DAG $G'$ (Line 15), if out-neighbor $v'_o$ has been visited (Line 16, case 1) and is in $aff$ (Line 17), but $v'_o$ is not in $aff$ yet, then we add $v'_o$ into $aff$ and add the edge into $E'_{nec}$. Case 1 is designed to facilitate Lemma 5 presented later. If $v'_f$ is not visited, then procedure MergeMSC is invoked for $v'_f$ (Line 21 case 2), after which, $R$ is set to true and $v'_f$ is added into $aff$ and $E'_{nec}$ is updated accordingly (Lines 22-23).

Example 4.3. Fig. 5 shows an example of $MSC^d$, with new edge in blue $(v'_1, v'_2)$. The number on each edge represents the DFS order by Algo. 6. The red edges are the necessary edges in $E'_{nec}$ after applying the algorithm. In the first two steps, we find a path $(v'_1, v'_2, v'_3)$, indicating the MSCSs that need to merge in $G$. Consequently, edges in this path are added into $E'_{nec}$, and $v'_1, v'_2$ and $v'_3$ are added into $aff$. Then, we find nodes and locate necessary edges in other paths from $v'_1$ to $v'_3$. A path $(v'_1, v'_2, v'_3)$ is found where $v'_2$ is not in $aff$. Then we add $v'_2$ into $aff$ and edges in this path become necessary. When we reach $(v'_3, v'_4)$, since $v'_2$ and $v'_3$ are both in the $aff$, then this edge is unnecessary. We can see that the nodes in $aff$, $(v'_1, v'_2, v'_3, v'_4, v'_5)$, represent the MSCSs to merge, and the necessary edges in $G'$ are $E'_{nec} = \{(v'_1, v'_2), (v'_2, v'_3), (v'_1, v'_3), (v'_2, v'_3), (v'_4, v'_5)\}$. For each edge $e' \in E'_{nec}$, we choose an arbitrary edge $e$ in $G$ that maps to $e'$ and add $e$ to the updated $E_{nec}$.

Analysis of $MSC^d$. In Lemma 5, we prove that Algo. 6 finds a 2-approximate MSCSC $E_{nec}$ of the SCC formed in $G'$ with a new edge. Then Theorem 4.4 states the 2-approximation guarantee of Algo. 6, for the updated $E_{nec}$ obtained for graph $G$ with a new edge insertion. The time and space complexities of $MSC^d$ are both $O(n' + m')$ as it

Algorithm 8: SplitMSCS

**Input:** $G$, deleted edge $(u_d, v_d)$, low, depth, $S$, $G_S$, $E_{tree}$, $E_{lastdrop}$, redo, $u$

**Output:** new necessary edges

1. if $u = u_d$ then
2. \hspace{1cm} if $|E_{nec}| > 2|V(G_S)| - 2$ then
3. \hspace{2cm} redo $\leftarrow$ true
4. \hspace{1cm} return false
5. \hspace{1cm} $low(u) \leftarrow depth$, $dfn(u) \leftarrow depth$, $depth \leftarrow depth + 1$
6. stack $S.push(u)$, visited[$u$] $\leftarrow$ true
7. $E_{lastdrop} \leftarrow \emptyset$
8. for each out-going edge $(u, v)$ of $u$ in $G_S$ do
9. \hspace{1cm} $E_{lastdrop}.remove((u, v))$, $E_{tree}.remove((u, v))$
10. \hspace{1cm} if visited[$v$] = false then \hspace{1cm} // case 1
11. \hspace{2cm} $E_{tree}.add((u, v))$
12. \hspace{1cm} if SplitMSCS($v$) = false and redo = false then
13. \hspace{2cm} \hspace{1cm} return false
14. \hspace{1cm} if $low(u) \geq low(v)$ then
15. \hspace{2cm} \hspace{1cm} $E_{lastdrop} \leftarrow ((u, v))$
16. \hspace{1cm} \hspace{1cm} $low(u) \leftarrow low(v)$
17. \hspace{1cm} else if $v \in Stack$ and $low(u) > dfn(v)$ then \hspace{1cm} // case 2
18. \hspace{2cm} \hspace{1cm} $E_{lastdrop} \leftarrow ((u, v))$
19. \hspace{2cm} \hspace{1cm} $low(u) \leftarrow dfn(v)$
20. \hspace{1cm} \hspace{1cm} if $E_{lastdrop} \neq \emptyset$ then
21. \hspace{2cm} \hspace{2cm} $E_{lastdrop}.add(E_{lastdrop})$
22. \hspace{1cm} \hspace{1cm} Repeat Lines 17-19 in Algo. 1 to create SCCs
23. \hspace{1cm} return true

visits every edge in $G'$ at most once (procedure MergeMSCS) with constant cost per edge, where $n'$ and $m'$ are the number of nodes and edges in $G'$.

**Lemma 5**. In the DAG $G'$, suppose that there are cycles formed after inserting an edge $(u', v')$, the necessary edge set $E_{nec}$ of $G'$ returned by Algo. 6 is 2-approximate.

**Theorem 4.4**. Given a graph $G$ with 2-approximate MSCSC $E_{nec}$, after inserting an edge, the updated $E_{nec}$ by Algo. 6 is 2-approximate.

4.2 Decremental Update

When deleting edge $(u_d, v_d)$ in graph $G$, obviously, approximate MSCSC $E_{nec}$ is affected only when the edge is in $E_{nec}$. If $u_d$ and $v_d$ are from different MSCSs, or $u_d$ and $v_d$ are in the same MSCS but $(u_d, v_d)$ is not in $E_{nec}$, then nothing needs to be done to maintain $E_{nec}$. If edge $(u_d, v_d)$ is in $E_{nec}$ (i.e., $u_d$ and $v_d$ are in the same MSCS), the deletion may cause the split of the MSCS. However, if we can find another path in $G$ from $u_d$ to $v_d$ and the path does not contain edge $(u_d, v_d)$, then the MSCS does not need to split and we only need to update the relevant necessary edges in the path in $E_{nec}$. If there exists no path from $u_d$ to $v_d$ after deleting the edge in $G$, then the MSCS splits, and we need to identify the resulted new MSCSs.

A naive method is to invoke Algo. 3 to find new MSCSs and update necessary edges inside the MSCS, with which $E_{nec}$ is a 2-approximate. However, this method is inefficient since we can actually terminate immediately when another path from $u_d$ to $v_d$
We find the new MSCSs and return at Lines 22-23. Whenever a path we initialize and set the low value. Therefore, we present a decremental update method for MSCs. Algorithm 4 is invoked to determine if the MSCS splits or not and update necessary edges. If there is a split, we need to detect the new MSCs by Algorithm 4 for every node $u$ to get them (Lines 8-10). Finally, $E_{new}$ is updated at Line 11.

As mentioned, procedure SplitMSCS (Algorithm 8) determines if the MSCS splits or not (i.e., if there is another path from $u_q$ to $v_q$) and updates necessary edges simultaneously. If such a path is found, then SplitMSCS marks edges in this path as necessary to keep the connectivity from $u_q$ to $v_q$ and returns immediately, to save computational costs (Lines 1-4). If no such path is found, from Lines 5 to 19, we continue the traversal and make this decremental procedure perform like conducting Algorithm 3. To tackle both scenarios simultaneously, a vital step in SplitMSCS different from Algorithm 3 is that we must mark a newly visited edge as unnecessary whenever we reach it, and then decide if it is necessary later on. Specifically, we initialize and set the $low$, $dfn$, depth values, as well as stack $S$ and visited flags, at Line 5-6. Then for every out-neighbor $v$ of node $u$ in $G_S$, we remove the edge from $E_{lastdrop}$ and $E_{tree}$ first at Line 9, and will decide to add it back or not later at Lines 11, 15, and 18. Lines 10-19 are similar to Algorithm 4, except Lines 12-13, where it recursively decides if no split occurs and redo is necessary or not. We find the new MSCSs and return at Lines 22-23. Whenever a path from $u_q$ is found at Line 1, to ensure 2-approximation, we verify if the number of edges in $E_{new}$ exceeds $2|V(G_S)| - 2$, the maximum possible number of necessary edges for 2-approximation in $G_S$, at Lines 5-7. If yes, we set the redo flag to be true, which will lead to the execution of Lines 8-10 in Algorithm 7 to get new $E_{new}$.

Example 4.5. Fig. 6 shows an example of decremental necessary edge maintenance. The red edges indicate the necessary edges in $E_{new}$. Supposing that $(v_1, v_2)$ is deleted at Line 6(a) is deleted. Since this edge is a necessary edge, we need to check whether there is an alternative path from $v_1$ to $v_2$ in the updated graph as shown in Fig. 6(b). A path $(v_3, v_4, v_1, v_2)$ can be found, which indicates that this MSCS will not split. Then to maintain the connectivity of vertices in this path, edges $(v_3, v_4), (v_4, v_1), (v_1, v_2)$ in this path are added into $E_{new}$ (Fig. 6(b)).

### Table 2: Statistics of Datasets. ($K = 10^3, M = 10^7, B = 10^5$)

<table>
<thead>
<tr>
<th>Name</th>
<th>Dataset</th>
<th>V</th>
<th>E</th>
<th>$d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>EP</td>
<td>Epinions</td>
<td>75.9K</td>
<td>509R</td>
<td>6.7</td>
</tr>
<tr>
<td>YT</td>
<td>Youtube</td>
<td>1.14M</td>
<td>4.94M</td>
<td>4.3</td>
</tr>
<tr>
<td>IN</td>
<td>IN-2004</td>
<td>1.38M</td>
<td>16.5M</td>
<td>12</td>
</tr>
<tr>
<td>WF</td>
<td>Wikitr</td>
<td>3.33M</td>
<td>124M</td>
<td>37.1</td>
</tr>
<tr>
<td>EU</td>
<td>EU-2005</td>
<td>11.3M</td>
<td>360M</td>
<td>33.7</td>
</tr>
<tr>
<td>FT</td>
<td>TW-2004</td>
<td>41.3M</td>
<td>1.14B</td>
<td>27.5</td>
</tr>
<tr>
<td>T3W</td>
<td>TwitterW/W</td>
<td>41.7M</td>
<td>1.47B</td>
<td>35.3</td>
</tr>
<tr>
<td>FS</td>
<td>Friendster</td>
<td>68.3M</td>
<td>2.59B</td>
<td>37.8</td>
</tr>
</tbody>
</table>

Then, we terminate traversal without visiting $(v_1, v_3)$ and $(v_2, v_4)$. The final necessary edges are in red in Fig. 6(b).

### Correctness and Complexity Analysis

If the deleted edge $(u_q, v_q)$ is not in $E_{new}$, then nothing needs to do. Otherwise, there are two cases. The first case is that the corresponding MSCS splits. In this case, $M_{SC}$ performs like Algorithm 3 inside the induced graph $G_S$. The second case is that the MSCS will not split, and we find another path from $u_q$ to $v_q$ and insert the edges on the path into $E_{new}$, which maintains the connectivity from $u_q$ to $v_q$. Thus, $E_{new}$ updated by Algorithm 7 maintains the strong connectivity of $G$. The time and space complexities of Algorithm 7 are both $O(|V(G_S)| + |E(G_S)|)$. It visits every edge in $G_S$ at most once by procedure SplitMSCS with constant per-edge cost. Since $G_S$ is a subgraph of $G$, the complexity is rewritten as $O(n + m)$. Additionally, it can terminate early once the deleted edge is not necessary (Lines 2-3 in Algorithm 7), or the MSCS will not split (Lines 1-4 and 12-13 in Algorithm 8). As a result, its practical performance is better than its worst-case complexity.

**Theorem 4.6.** Given a graph $G$ with 2-approximate MSCSC $E_{new}$ after deleting an edge, the updated $E_{new}$ by Algorithm 7 is 2-approximate.

### 5 EXPERIMENTS

We conduct experiments on a Linux machine with an Intel Xeon 2.10GHz CPU and 504GB memory. All algorithms are in C++ and compiled via g++ with full optimization. Our code is at [1].

#### 5.1 Experimental Setup

**Datasets.** We test on 8 real graph datasets with statistics in Tab. 2. All datasets are publicly available from SNAP [29], KONECT [27], and WebGraph [8]. IT, T3W, and FS contain billions of edges, and FS is the largest directed graph available in KONECT [27]. For each graph, we remove self-loops and multi-edges.

**Competitors.** Zhao [55] is a linear-time MSCS method, while the other methods [25, 26, 48] run in super-linear time. Kuhl1er [25] runs in a near-linear time and is 7/4-approximate. Therefore, we extend Kuhl1er and Zhao to MSCSC. For static graphs, Kuhl1er and Zhao first apply Algorithm 1 to detect SCCs and then detect MSCS from each SCC. For dynamic graphs, Kuhl1erDyn and ZhaoDyn first identify if MSCS split or merge happens, and then update MSCSs only when necessary. A method will be terminated after running 24 hours without returning results, i.e. OOT.

**Evaluation Metrics.** For approximation performance, since the ground truth is hard to obtain, we calculate a necessary ratio $R_{new} = |E_{new}|/|edges in SCCs|$, i.e., the ratio of the number of edges in approximate $E_{new}$ over the number of all edges in SCCs of $G$. A lower necessary ratio $R_{new}$ indicates a tighter approximation.
5.2 MSCSC Evaluation

We evaluate the performance under three workloads: edge deletion, edge insertion, and mixed workload. We also report the MSCSC construction performance and the scalability on synthetic graphs.

**Edge Deletion.** Given a graph $G$, we select 10K edges uniformly at random and delete them from $G$. For every edge deletion, we run a method to update the MSCSC $E_{\text{rec}}$. Fig. 7(a) reports the average MSCSC maintenance time in milliseconds (ms) on all edge deletions over all datasets of our method $\text{MSC}^d$, Khuller$_{dyn}$, and Zhao$_{dyn}$. Observe that $\text{MSC}^d$ is consistently faster than Zhao$_{dyn}$ and Khuller$_{dyn}$, often by an order of magnitude. For instance, on T3W, a large graph with billions of edges, $\text{MSC}^d$ updates MSCSC in 960 ms per edge deletion, which is 10 times faster than Zhao$_{dyn}$ that costs 9200 ms and 20 times faster than Khuller$_{dyn}$ that takes 17 800 ms. Moreover, on the largest FS graph, $\text{MSC}^d$ is efficient, while Khuller$_{dyn}$ and Zhao$_{dyn}$ run OOT. The speedup of $\text{MSC}^d$ over the competitors validates the efficiency of the techniques proposed in Section 4.2 for dynamic MSCSC under edge deletion. $\text{MSC}^d$ only needs to focus on the local subgraph affected and scans the edges in the subgraph only once, while Khuller$_{dyn}$ and Zhao$_{dyn}$ need to compute from scratch and scan the subgraph twice. Further, in the second column of Table 3, after massive edge deletions, the necessary ratios of $\text{MSC}^d$ remain stable on all datasets with a negligible increase compared with Khuller$_{dyn}$ and Zhao$_{dyn}$, validating the effectiveness of our techniques for dynamic MSCSC under edge deletions, and indicating the better trade-off for efficiency achieved by $\text{MSC}^d$ in Fig. 7(a).

**Edge Insertion.** We then regard the deleted edges above as new edges to insert back into the graph, and evaluate the efficiency of MSCSC maintenance under edge insertions as reported in Fig. 7(b). Observe that $\text{MSC}^i$ consistently outperforms Zhao$_{dyn}$ and Khuller$_{dyn}$ by a significant margin, often in orders of magnitude. On IT with 1.14 billion edges, $\text{MSC}^i$ runs in 2 ms to maintain MSCSC per edge insertion, while Zhao$_{dyn}$ requires 370 ms, which is 135 times slower and Khuller$_{dyn}$ requires 836 ms, which is 418 times slower. On the largest FS graph with 68.3 million nodes and 2.59 billion edges, Khuller$_{dyn}$ and Zhao$_{dyn}$ run OOT. For edge insertions, $\text{MSC}^i$ works on the reduced DAG $G'$ to identify the MSCs that need to merge and then update $E_{\text{rec}}$ in $G$ accordingly, which explains its superiority compared with Khuller$_{dyn}$ and Zhao$_{dyn}$. Tab. 3 shows the necessary ratios $R_{\text{rec}}$ of $\text{MSC}^i$ that is close to Khuller$_{dyn}$ and Zhao$_{dyn}$, which validates the effectiveness of our techniques in Section 4.1 for dynamic MSCSC under edge insertions.

**Mixed Workload.** In a mixed workload, for every graph, we randomly generate 10K edge deletions, and also randomly generate 10K edge insertions (we delete these edges from the graph before the update starts), and then obtain the mixed workload with 20K edge updates by combining and randomly shuffling the 10K edge deletions and 10K edge insertions. For our method ($\text{MSC}^i + \text{MSC}^d$), Fig. 7(c) shows the average update time and the last column in Tab. 3 reports $R_{\text{rec}}$ under the mixed workload. In Fig. 7(c), $\text{MSC}^i + \text{MSC}^d$ is 6X-7X faster than Zhao$_{dyn}$ in six datasets (EP, YT, IN, WF, EU, and IT), and one order of magnitude faster in T3W, and Zhao$_{dyn}$ runs OOT in FS, while Khuller$_{dyn}$ is even slower. Moreover, in Tab. 3, observe that $R_{\text{rec}}$ of all methods remain close to each other on all datasets. Hence, we conclude that we achieve a better trade-off between efficiency and effectiveness.

**MSCSC Construction Time and Approximate Ratio.** We evaluate the efficiency of MSC in Algo. 3, Khuller and Zhao to build MSCSC $E_{\text{rec}}$ (i.e., efficiency on static graphs), and compare their practical approximation performance. The second, third, and fourth columns of Tab. 4 report the construction time of MSC, Khuller, and Zhao on all datasets. MSC is nearly 3 times faster than Zhao on most datasets and 5 times faster than Zhao on FS, since our method only needs to traverse each edge once. Further, MSC is much faster than Khuller, e.g., almost 10 times faster on T3W dataset. The last three columns of Tab. 4 report the necessary ratio $R_{\text{rec}}$. Observe that $R_{\text{rec}}$ of MSC is close to that of Khuller and Zhao, indicating that...
our method provides close practical approximation, which certifies the small theoretical guarantee gap among these methods.

**Comparison on MSC\textsuperscript{1+} in Algo. 5 and MSC\textsuperscript{1} in Algo. 6.** Recall that in Sec. 4.1, we first develop an optimal solution for incremental MSCSC maintenance (Algo. 5), and then present a more practical solution (Algo. 6). We use the same 10K edge insertions above for evaluation and report the average runtime in Tab. 5 as well as the differences (Δ) in the number of edges in their respective MSCSC solutions after handling all edge insertions. The observation is that MSC\textsuperscript{1} yields higher efficiency with a significant speedup ratio over MSC\textsuperscript{1+}. The number of edges in $\mathcal{E}_{\text{rec}}$ of the optimal incremental solution MSC\textsuperscript{1+} is always smaller than that of MSC\textsuperscript{1}, but they are close with small Δ values, indicating that MSC\textsuperscript{1} in Algo. 6 is practically effective in maintaining tight $\mathcal{E}_{\text{rec}}$, while being much more efficient.

**Scalability.** We vary graph size and density to evaluate the scalability of our approach. To vary graph size, we generate random graphs using the generator in [53] with the number of nodes in {1M, 2M, 4M, 8M, 16M}, while keeping average node degree as 16, so as to scale the number of edges proportional to the number of nodes. Then, on each graph, we run MSC\textsuperscript{1} + MSC\textsuperscript{d} to handle a mixed workload with 10K edge insertions and 10K edge deletions, and report the average update time and necessary ratio $R_{\text{rec}}$ in Fig. 8(a). Observe that the running time increases, since there are more nodes to handle and more necessary edges to detect. Meanwhile, as the number of nodes doubles, the number of necessary edges and edges in SCCs both increase proportionally and thus, $R_{\text{rec}}$ remains relatively stable. To vary graph density, we generate graphs with average node degree in [4, 8, 16, 32, 64], while keeping the number of nodes as $n = 1M$. Then we run MSC\textsuperscript{1} + MSC\textsuperscript{d} on the mixed workloads of these graphs and report running time and $R_{\text{rec}}$ in Fig. 8(b). Observe that $R_{\text{rec}}$ decreases as density increases. With higher graph density, the number of edges in $\mathcal{E}_{\text{rec}}$ remains relatively stable and is bounded by $2n \sim 2$, but the number of edges in SCCs increases, resulting in the decrease of $R_{\text{rec}}$. Specifically, when the average degree varies from 4 to 64, the number of edges in $\mathcal{E}_{\text{rec}}$ is [1.18M, 1.24M, 1.21M, 1.16M, 1.12M], while the number of edges in SCCs is [3.64M, 7.86M, 15.9M, 32M, 64M]. Running time also reduces as graph density increases. With higher density, more edges are redundant for strong connectivity. That is, in a denser graph, more edges to be deleted are not in $\mathcal{E}_{\text{rec}}$, and nothing needs to be done. Similarly, in a denser graph, edge insertions may happen between nodes in the same MSCS, and $\mathcal{E}_{\text{rec}}$ does not need to be updated.

### 5.3 Use Case Studies

We present two use cases to demonstrate that our MSCSC methods can readily speed up dynamic SCC maintenance and dynamic reachability index maintenance, which are two important processing tasks in graph systems [32, 38], revealing the potential of our methods to be adopted into these systems.

**Use Case 1: Applying MSCSC for Fully Dynamic SCC Maintenance.** We apply our MSCSC solutions to improve the efficiency of fully dynamic SCC maintenance under edge insertions and deletions. Existing studies for dynamic SCC maintenance mainly focus on reducing the theoretical bound on time complexity. Given a graph with $n$ nodes, a recent method Adam\textsubscript{SCC} [24] theoretically achieves state-of-the-art worst-case time complexity $O(n^2 \log^2 n)$, which however is not practical with immense memory consumption. In experiments, Adam\textsubscript{SCC} runs out of memory (OOM) even for the smallest data EP. To achieve the time complexity, Adam\textsubscript{SCC} needs to create more than $4 \cdot \log^3 n$ copies of the input graph, e.g., more than 106K copies of EP (with 54 billion edges in total). Then, we choose to compare with the SCC maintenance method in the paper of DAGGER [53], which can scale to large graphs. With a mixed workload of 10K edge insertions and 10K edge deletions on each graph, we report the running time in Fig. 9. Our method (MSC\textsuperscript{1} + MSC\textsuperscript{d}) consistently achieves higher efficiency than the competitor in terms of average update time, specifically, 2X-3X faster in EP, YT and IN, 4X faster in EU and T3W, and 6X-7X faster in WF and IT, and the competitor runs OOT on FS. The results show that our method can significantly accelerate fully dynamic SCC maintenance.

**Use Case 2: Applying MSCSC to Dynamic Reachability Index Maintenance.** We apply our MSCSC solutions to an important use case: improving the efficiency for maintaining dynamic SCC-based reachability index maintenance, such as TOL [56] and DAGGER [53], but not for non-SCC reachability methods [33, 39, 40]. Specifically, TOL refers to the Total Ordering Labeling (TOL) framework [56] that works on the corresponding DAG $G'$ reduced from the input graph $G$, either by

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Time of MSC\textsuperscript{1}</th>
<th>Time of MSC\textsuperscript{1+}</th>
<th>Speedup  90%</th>
<th>Speedup  100%</th>
<th>Speedup  110%</th>
<th>Speedup  120%</th>
</tr>
</thead>
<tbody>
<tr>
<td>EP</td>
<td>0.0262</td>
<td>0.103</td>
<td>3.94</td>
<td>5.24</td>
<td>6.94</td>
<td>8.82</td>
</tr>
<tr>
<td>YT</td>
<td>1.9</td>
<td>15.3</td>
<td>8.1</td>
<td>13.7</td>
<td>20.6</td>
<td>27.5</td>
</tr>
<tr>
<td>IN</td>
<td>0.173</td>
<td>1.4</td>
<td>8.1</td>
<td>11.4</td>
<td>16.2</td>
<td>21.1</td>
</tr>
<tr>
<td>WF</td>
<td>0.0274</td>
<td>0.167</td>
<td>6.1</td>
<td>8.4</td>
<td>12.3</td>
<td>16.2</td>
</tr>
<tr>
<td>EU</td>
<td>2.68</td>
<td>19.1</td>
<td>7.1</td>
<td>11.6</td>
<td>17.1</td>
<td>22.6</td>
</tr>
<tr>
<td>IT</td>
<td>1.99</td>
<td>21.1</td>
<td>10.6</td>
<td>15.3</td>
<td>21.0</td>
<td>26.7</td>
</tr>
<tr>
<td>T3W</td>
<td>3.85</td>
<td>15.3</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>FS</td>
<td>13.8</td>
<td>79.4</td>
<td>5.8</td>
<td>8.5</td>
<td>11.1</td>
<td>13.4</td>
</tr>
</tbody>
</table>
MSCs or SCCs. Since TOL only supports vertex insertion/deletion, we extend it into supporting edge insertion/deletion. In particular, TOL+MSCSC adopts our dynamic MSCSC solutions and builds a 2-hop index for dynamic reachability query processing. TOL+SCC adopts dynamic SCCs and the same 2-hop index, and DAGGER is an existing dynamic solution for reachability queries. We also compare with DBL [33] that is a recent dynamic non-SCC reachability index on general graphs, and IP [50] that is a dynamic randomness-based reachability index. Note that DBL only supports edge insertions, and we extend it to support edge deletions; IP is designed for DAG, and we extend its capability to handle general graphs.

To evaluate the dynamic maintenance efficiency of reachability indices, we employ the same mixed workload in Section 5.2. Fig. 10 shows the average time to maintain reachability indices per update in milliseconds. We can observe that TOL+MSCSC is at least two orders of magnitude faster than DBL, IP, and DAGGER, and these three competitors run out of time after 24 hours on WF, EU, IT, T3W, and FS. Compared with TOL+SCC, TOL+MSCSC is nearly 2X faster in EP and YT, 3X faster in IN and FS, 5X faster in EU and T3W, and 8X faster in WF and IT. TOL+MSCSC only maintains necessary edges $E_{nee}$ instead of every edge in an SCC as TOL+SCC does, and thus, TOL+MSCSC is more efficient. To evaluate query time, we follow the setting in [50] to randomly generate 10K queries and calculate the average query time of every method on every dataset. Table 6 reports the query time results. We can conclude that TOL+MSCSC is six orders of magnitude faster than DAGGER as TOL+MSCSC adopts the 2-hop index to accelerate the query processing. The query time of TOL+MSCSC and TOL+SCC is similar to each other since both of them build the same 2-hop index in the reduced graph. Note that our focus is on the efficiency of dynamic reachability index maintenance, rather than query efficiency. IP has similar query performance as TOL, and DBL has competitive query performance on EP and WF, while being worse on other datasets.

7 CONCLUSION

We propose a new problem MSCSC to find a collection of subgraphs, each of which is maximal in terms of nodes and are strongly connected via the fewest edges. We develop efficient approximate solutions for both static and dynamic graphs. In particular, we first present MSC which is a static MSCSC method and performs only one scan of graph $G$ with linear time complexity to get approximate MSCSC with rigorous approximation guarantees. We then develop efficient MSC$^d$ and MSC$^{cd}$ to maintain dynamic MSCSC with edge insertions and deletions, respectively. Extensive experiments and use cases validate the high efficiency of our methods on large-scale graphs. Our future work is to consider property graphs with properties on nodes and edges to formulate a property-constrained MSCSC problem. We will investigate how to extend the proposed techniques to handle such property graphs.

ACKNOWLEDGMENTS

This work was supported by Hong Kong RGC GRF (Grant No. 14217322), Hong Kong ITC ITF (Grant No.MRP/071/20X), Hong Kong RGC ECS (No. 25201221), NSFC 62202404, and ARC Future Fellowship FT210100303.