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
edwlin@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 `MSC` 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 `MSC`, we leverage a reduced directed acyclic graph of G to design incremental MSCSC method $\texttt{MSC}^{\texttt{i}}$ with two variants to handle edge insertions efficiently. We further develop $\texttt{MSC}^{\texttt{d}}$ 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:
Xin Chen, Jieming Shi, You Peng, Wenqing Lin, Sibo Wang, and Wenjie Zhang. Minimum Strongly Connected Subgraph Collection in Dynamic Graphs. PVLDB, 17(6): 1324-1336, 2024.
doi:10.14778/3648160.3648173

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

*Jieming Shi is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 17, No. 6 ISSN 2150-8097.
doi:10.14778/3648160.3648173

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,

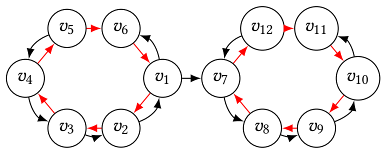


Figure 1: Red edges form the MSCSC containing two MSCSs

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_1 to v_6, 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 α-approximate MSCSC to find an edge set E_{nec} with size bounded by an approximation factor α on the size of the optimal solution in Section 2. Then, we develop a new 2-approximate MSCSC method `MSC` 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 $\texttt{MSC}^{\texttt{i}}$ that only works on the locally affected subgraphs for MSCSC updates under new edge insertions. In particular, we develop two variants of $\texttt{MSC}^{\texttt{i}}$ 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 $\texttt{MSC}^{\texttt{d}}$ 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` that runs one scan of the graph to identify approximate MSCSC with rigorous guarantees.
- We then present $\texttt{MSC}^{\texttt{i}}$ and $\texttt{MSC}^{\texttt{d}}$ 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.

Table 1: Frequently used notations.

Notation	Description
$G = (V, E)$	A directed graph G with vertex set V and edge set E
n, m	$n = \|V\|, m = \|E\|$
$G' = (V', E')$	A reduced graph G' with node set V' and edge set E'
$f(\cdot)$	The mapping function between G and G'
E_{nec}, E_{nec}^{opt}	An approximate MSCSC of G containing the necessary edges, and the optimal MSCSC solution
α	Approximation ratio
G_S, S	A strongly connected graph, and a strongly connected component
$\langle u, v\rangle$	A directed edge from u to v in G
$\langle u', v'\rangle$	A directed edge from u' to v' in G'

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}^{opt} be the set of edges in the optimal MSCSC solution for G.

Algorithm 1: TARJAN(G)

```
1  depth ← 1, visited[v] = false ∀ v ∈ V
2  Initialize a global stack S and set SCCs ← ∅
3  for each vertex u ∈ V do
4      if visited[u] = false then
5          DFS(u)
6  return SCCs
7
8  Procedure DFS(u)
9  low(u) ← depth, dfn(u) ← depth
10 S.push(u), visited[u] ← true, depth ← depth + 1
11 for each outgoing edge ⟨u, v⟩ of u do
12     if visited[v] = false then                          // case 1
13         DFS(v)
14         low(u) ← min{low(u), low(v)}
15     else if v ∈ S then                                  // case 2
16         low(u) ← min{low(u), dfn(v)}
17 if low(u) = dfn(u) then                                 // create an SCC
18     Pop all elements in stack S until it reaches u and add them to
         an SCC S. Add S to SCCs
19     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_{nec}^{opt} is the set of red edges. Intuitively, all edges in E_{nec}^{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_{nec}^{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_{nec}^{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_{nec}^{opt} of MSCSC in G. Hence, we focus on α-approximate MSCSC.

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

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_{nec}, dynamic MSCSC maintains the up-to-date E_{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.

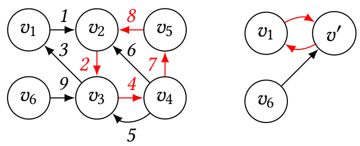
Tarjan's SCC algorithm [45]. There exist algorithms to efficiently find SCCs [14, 42, 45]. Tarjan's algorithm [45] is one representative method, with its pseudo-code in Algo. 1. The whole algorithm runs in a depth-first search (DFS) manner. Initially, at Line 1, it sets *depth* to be 1, which is a global 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 $\mathcal{S}$ 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 $dfn(\cdot)$ 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 $\mathcal{S}$. The main idea is that if a node v_{first} is the first node visited among all nodes in an SCC, then node v_{first} must have the smallest $dfn(v_{first})$ value among all nodes in the SCC, and $dfn(v_{first})$ also equals to $low(v_{first})$, while the other nodes v in the SCC are with $dfn(v) > low(v)$. The global stack $\mathcal{S}$ is used to find all nodes in the same SCC. In Tarjan's algorithm, for the nodes in the same SCC as v_{first}, they will be on top of v_{first} in the stack $\mathcal{S}$. Then, we can retrieve the SCC containing v_{first} via popping all elements in $\mathcal{S}$ until v_{first} 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 $low(u)$ value is updated if $low(v)$ is smaller (Line 14). Otherwise, v has already visited, and if v is already in $\mathcal{S}$ (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 $\mathcal{S}$ are popped out and added to S (Lines 18-19). The time and space complexities of Algo. 1 are both $O(n + m)$.

Algorithm 2: ZHAO(G)

```
Input: A strongly connected graph G
Output: A minimum strongly connected subgraph MSCS
G# ← G, E' ← ∅
while G# contains a concealing cycle C of length at least 3 do
    E' ← E' ∪ E(C), G# ← G# \ V(C)
E' ← E' ∪ E(G#)
return a new graph G* = (V, E')
```

Dynamic SCCs. In literature, there exist studies for SCC maintenance [5–7, 28, 37, 53]. 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



(a) G (b) G_z

Figure 2: Running Example of Zhao **Method**

contracts concealing cycles of length at least 3 until no such cycle exists (Lines 2-3). A cycle C in a graph G is a concealing cycle if there exists a node set $V' \subseteq V$ such that $\delta^+_G(V') \neq \emptyset$ and all edges in $\delta^+_G(V')$ will be removed by contracting C, where $\delta^+_G(V')$ is the set of outgoing edges of nodes in V'. Fig. 2 presents a running example of Zhao. Graph G contains two SCCs: $\{v_6\}$ and $\{v_1, v_2, v_3, v_4, v_5\}$. On the large SCC, to detect MSCS, 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 $\langle v_1, v'\rangle$ and $\langle v', v_1\rangle$ correspond to $\langle v_1, v_2\rangle$ and $\langle v_3, v_1\rangle$ in G, respectively. As there is no concealing cycle of length at least 3 in G_z, Zhao 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', and red edges inside this cycle become necessary. Then, Zhao finds the necessary edges for the SCC, including $\langle v_1, v_2\rangle$, $\langle v_2, v_3\rangle$, $\langle v_3, v_1\rangle$, $\langle v_3, v_4\rangle$, $\langle v_4, v_5\rangle$, and $\langle v_5, v_2\rangle$.

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 MSCS 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_{nec} 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_{nec} of a graph G.

For the ease of understanding, in the following, we focus on the analysis on a strongly connected graph G_S. 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_S, when performing DFS traversal from a visited node u, an edge $\langle u, v\rangle$ is a tree edge if u reaches an unvisited node v via edge $\langle u, v\rangle$ and u and v are strongly connected.

Further, as shown in Algo. 1, an edge $\langle u, v\rangle$ 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 $\langle u, v\rangle$ 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_S, we denote the edge $\langle u, v\rangle$ that causes the drop of $low(u)$ of node u as a dropping edge of u.

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

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

However, note that $E_{tree}(G_S) \cup E_{drop}(G_S)$ does not have an approximation guarantee with respect to the optimal solution of G_S, 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 $low(v_4)$ is changed from 4 to 3 (due to edge $\langle v_4, v_3\rangle$) and then to 2 (due to edge $\langle v_4, v_2\rangle$). 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 $E_{lastdrop}(G_S)$, 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 $\langle u, v\rangle$ 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_S via last dropping edges only $E_{lastdrop}(G_S)$. Then it is natural to derive Lemma 3 that $E_{tree}(G_S)$ and $E_{lastdrop}(G_S)$ together can preserve the strong connectivity of any two nodes in G_S.

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

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

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_{nec}, w.r.t., E^{opt}_{nec}.

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_{nec} be the union of all tree edges and last dropping edges in G. E_{nec} is a 2-approximation MSCSC w.r.t., the optimal, i.e., $|E_{nec}|/|E^{opt}_{nec}| \leq 2$.*

Algorithm 3: Approximate MSCSC: MSC

Input: Directed graph G
Output: Approximate MSCSC E_{nec}
1 $E_{tree} \leftarrow \emptyset, E_{lastdrop} \leftarrow \emptyset$
2 $depth \leftarrow 1, visited[v] \leftarrow false \ \forall\ v \in V$
3 **for each** *node* $u \in V$ **do**
4   **if** *visited[u]* $= false$ **then**
5         $ProcessNode(u)$
6 $E_{nec} \leftarrow E_{lastdrop} \cup \left(\bigcup_{\langle u,v \rangle \in E_{tree}, f(u)=f(v)} \langle u, v \rangle \right)$

Algorithm 4: ProcessNode

Input: G, $depth$, low, $\mathcal{S}$, E_{tree}, $E_{lastdrop}$, u
Output: last dropping edges, temporary tree edges, and MSCSs
1 $low(u) \leftarrow depth, dfn(u) \leftarrow depth, depth \leftarrow depth + 1$
2 Stack $\mathcal{S}.push(u)$, $visited[u] \leftarrow true$
3 $e_{lastdrop} \leftarrow \emptyset$
4 **for each** *outgoing edge* $\langle u, v \rangle$ *of* u **do**
5   **if** $visited[v] = false$ **then** // case 1
6         $E_{tree}.add(\langle u, v \rangle)$
7         $ProcessNode(v)$
8   **if** $low(u) \geq low(v)$ **then**
9             $e_{lastdrop} \leftarrow \langle u, v \rangle$
10             $low(u) \leftarrow low(v)$
11   **else if** $v \in \mathcal{S}$ *and* $low(u) > dfn(v)$ **then** // case 2
12         $e_{lastdrop} \leftarrow \langle u, v \rangle$
13         $low(u) \leftarrow dfn(v)$
14 **if** $e_{lastdrop} \neq \emptyset$ **then**
15     $E_{lastdrop}.add(e_{lastdrop})$
16 Repeat Lines 17-19 in Algo. 1 to create MSCSs instead of SCCs.

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 1-2 are with the same initialization as Algo. 1. At Line 3, $e_{lastdrop}$ 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 MSCS will be excluded from E_{tree} in the end at Line 6 of Algo. 3). After executing the procedure recursively for v at Line 7, if $low(u) \geq low(v)$ (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 $low(u) > dfn(v)$, this edge is updated as the last dropping edge $e_{lastdrop}$ of u. At the end of Algo. 4, it adds the last dropping edge into $E_{lastdrop}$ (Lines 14-15 of Algo. 4) and generates a new MSCS (Line 16 of Algo. 4). At the end of Algo. 3 (Line 6), it collects all necessary edges by first excluding false tree edges $\langle u, v \rangle$ not in the same MSCS (*i.e.*, $f(u) \neq f(v)$) from E_{tree}, and then taking the union of E_{tree} and $E_{lastdrop}$ (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 constant 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 $\langle v_1, v_2 \rangle$, $\langle v_2, v_3 \rangle$ and $\langle v_3, v_1 \rangle$. $\langle v_1, v_2 \rangle$ and $\langle v_2, v_3 \rangle$ are added to E_{tree} as tree edges, while $\langle v_3, v_1 \rangle$ is a dropping edge since $low(3)$ is changed from 3 to 1. It is now marked as the last dropping edge of v_3. In Fig. 3(b), we visit $\langle v_3, v_4 \rangle$ and $\langle v_4, v_3 \rangle$. Edge $\langle v_3, v_4 \rangle$ is added to E_{tree} as it is a tree edge. When we reach $\langle v_4, v_3 \rangle$, $low(v_4)$ is dropped and thus it is a dropping edge of v_4. We set $\langle v_4, v_3 \rangle$ temporarily as the last dropping edge of v_4. In Fig. 3(c), we deal with $\langle v_4, v_2 \rangle$. We will prune the previously stored last dropping edge $\langle v_4, v_3 \rangle$ since $low(v_4)$ is now updated again and edge $\langle v_4, v_2 \rangle$ becomes the new last dropping edge. After processing $\langle v_4, v_5 \rangle$ and $\langle v_5, v_2 \rangle$ in Fig. 3(d), $\langle v_4, v_5 \rangle$ is added to E_{tree} and $\langle v_5, v_2 \rangle$ is set as the last dropping edge of v_5 since $low(v_5)$ is changed. Also, the last dropping edge of v_4 is updated from $\langle v_4, v_2 \rangle$ to $\langle v_4, v_5 \rangle$ since $low(v_4) = low(v_5)$ and $\langle v_4, v_5 \rangle$ is a tree edge. Finally, a new MSCS is formed by v_1, v_2, v_3, v_4, and v_5. Also, v_6 forms another MSCS. And, we have $E_{tree} = \{\langle v_1, v_2 \rangle, \langle v_2, v_3 \rangle, \langle v_3, v_4 \rangle, \langle v_4, v_5 \rangle\}$. The set $E_{lastdrop}$ of last dropping edges is $\{\langle v_2, v_3 \rangle, \langle v_3, v_1 \rangle, \langle v_4, v_5 \rangle, \langle v_5, v_2 \rangle\}$. Unioning E_{tree} and $E_{lastdrop}$, we get E_{nec} (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_{nec} 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_{nec} (*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 MSCS of u' to any node in the MSCS of v' in G.

Our incremental methods first work on G' and then map back on G to maintain the approximate MSCSC E_{nec}. Given a new edge $\langle u_i, v_i \rangle$ 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_{nec} does not change, since u_i and v_i are already strongly connected via E_{nec}. If u_i and v_i belong to the 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

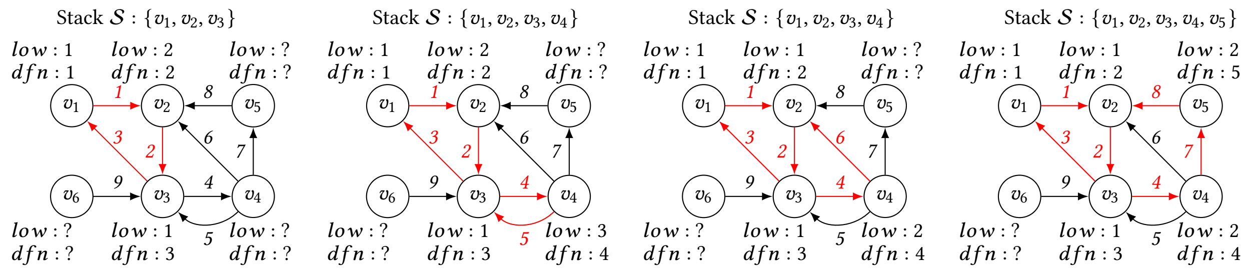


Figure 3: Running example for our method MSC.

of a graph G. If a new edge $\langle u_i, v_i\rangle$ is inserted into G and $f(u_i) = v_5', f(v_i) = v_1'$ in G' (*i.e.*, u_i and v_i are in different MSCSs), then it means that a corresponding edge in blue $\langle v_5', v_1'\rangle$ is also inserted into G'. Observe that the edge may cause the merge of the MSCSs. We first identify the strongly connected nodes in G', and consequently obtain the MSCSs that should be merged in G, and finally update E_{nec} accordingly in G, after which, a new DAG G' is also obtained.

Optimal 2-Approx Incremental MSCSC $\texttt{MSC}^{\texttt{i}*}$. $\texttt{MSC}^{\texttt{i}*}$ is optimal in the sense that, given a new edge insertion, the number of edges added in E_{nec} 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 $\langle u', v'\rangle$, where $f(u_i) = u'$, $f(v_i) = v'$ and $\langle u_i, v_i\rangle$ is the new edge in G. Then denote $G^* = S' \setminus \langle u', v'\rangle$. Apparently G^* is a DAG. For instance, in Fig. 4, DAG G' and the new edge in blue form an SCC. In this example, G^* is G' itself without the new edge. In DAG G^*, only node v' (resp. u') is with zero in-degree (resp. out-degree), e.g., v_4' and v_5' 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_{nec}' of G^*. For every edge $e' \in E_{nec}'$ 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_{nec}.

Algo. 5 presents the pseudo code of $\texttt{MSC}^{\texttt{i}*}$. The input includes the reduced DAG G', and a new edge $\langle u', v'\rangle$ (corresponding to a new edge $\langle u_i, v_i\rangle$ in graph G). The output is the updated E_{nec} 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 needs to be performed (Lines 3-4). Otherwise, we aim to identify the MSCS E_{nec}' 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 flag $reachU_i$ to indicate if it has any out-going edge added into E_{nec}' (*i.e.*, marked as necessary), which is initialized as false (Line 11). Then at Line 12, we iterate every out-going neighbor v_k' of v_j' (*i.e.*, out-going edge) to check if the edge is the last-visited incoming edge of v_k' by decreasing the indegree count of v_k' (Line 13). If the count becomes zero, it means all incoming edges of v_k' have been traversed, and $\langle v_j', v_k'\rangle$ 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_{nec}', and mark both $reach[v_k']$ and $reachU_i$ as true. After inspecting all outgoing edges of v_j' (Lines 12- 18), if $reachU_i$ is still false (Line 19), it means that none of v_j''s

Algorithm 5: Optimal Incremental MSCSC: $\texttt{MSC}^{\texttt{i}*}$

Input: Graph G with approximate MSCSC E_{nec}, 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_{nec} 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 $\langle u', v'\rangle$
3 **if** *no new SCC* **then**
4   **return**
5 $G^* \leftarrow S' \setminus \langle u', v'\rangle$, $E_{nec}' \leftarrow \{\langle u', v'\rangle\}$
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_j' from Q
11     $reachU_i \leftarrow false$
12   **for each** *outgoing edge $\langle v_j', v_k'\rangle$ of v_j' in G^** **do**
13         $d_{in}[v_k'] \leftarrow d_{in}[v_k'] - 1$
14   **if** $d_{in}[v_k'] = 0$ **then**
15             $Q.push(v_k')$
16   **if** $reach[v_k'] = false$ **then**
17                 $reachU_i \leftarrow true$
18                 $reach[v_k'] \leftarrow true$, $E_{nec}'.add(\langle v_j', v_k'\rangle)$
19   **if** $reachU_i = false$ **then**
20   let $\langle v_j', v_k'\rangle$ be one of outgoing edges of v_j'
21         $reach[v_k'] \leftarrow true$, $E_{nec}'.add(\langle v_j', v_k'\rangle)$
22 Produce a new G' by shrinking S' into a node
23 **for each** *edge $e' \in E_{nec}'$* **do**
24   Add one of edges e in G that maps to e' into E_{nec}

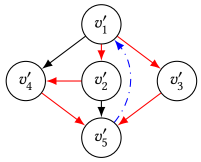


Figure 4: DAG G'

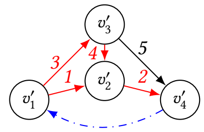


Figure 5: DAG G'; new edge in blue.

outgoing edges is added into E'_{nec}, then we just pick one outgoing edge and added it to E'_{nec} as well as marking the corresponding out-neighbor 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_{nec}, we choose one edge e in G that maps to e' and insert e into E_{nec}, 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 $f(u_i) = v'_5$ and $f(v_i) = v'_1$ in Fig. 4. Then, we add edge $\langle v'_5, v'_1 \rangle$ (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 v'_1 on G^*, since v'_1 is the only node with $d_{in} = 0$. Initially, only v'_1 is in Q. Then, we pop v'_1 from Q and update d_{in} of v'_4, v'_2 and v'_3. We find that $d_{in}[v'_2]$ (resp. $d_{in}[v'_3]$) becomes zero. As v'_2 (resp. v'_3) has only one incoming edge and $reach[v'_2]$ (resp. $reach[v'_3]$) is $false$, $\langle v'_1, v'_2 \rangle$ (resp. $\langle v'_1, v'_3 \rangle$) becomes necessary and $reach[v'_2]$ (resp. $reach[v'_3]$) becomes true. Also, we push v'_2 (resp. v'_3) into Q. Now, since $reachU_i$ of v'_1 becomes true, we can start the next iteration. Next, we pop v'_2 from Q and update d_{in} of v'_4 and v'_5. Now $d_{in}[v'_4] = 0$. Since $\langle v'_2, v'_4 \rangle$ is the only incoming edge of v'_4 and $reach[v'_4] = false$, this edge becomes necessary and $reach[v'_4]$ becomes true. Also, we push v'_4 into Q. Besides, since $reachU_i$ of v'_2 becomes true, we can turn to the next iteration. Next, we pop v'_3 from Q and update $d_{in}[v'_5]$ which is not zero. We find that $reachU_i$ of v'_3 is still false. Then we mark one arbitrary outgoing edge, say $\langle v'_3, v'_5 \rangle$ as necessary and $reach[v'_5]$ becomes true. Then, we pop v'_4 from Q and update $d_{in}[v'_5]$. We find that $d_{in}[v'_5]$ becomes zero. Though $d_{in}[v'_5] = 0$, since $reach[v'_5]$ is true, we will not immediately mark this edge as necessary in Line 14. Yet, since we find that $reachU_i$ of v'_4 is still false, we mark an outgoing edge $\langle v'_4, v'_5 \rangle$ as necessary. Also, we push v'_5 into Q. At last, the topological sort ends with popping v'_5 from Q. Finally, we get $E'_{nec} = \{\langle v'_1, v'_2 \rangle, \langle v'_2, v'_4 \rangle, \langle v'_4, v'_5 \rangle, \langle v'_1, v'_3 \rangle, \langle v'_3, v'_5 \rangle, \langle v'_5, v'_1 \rangle\}$. For each edge $e' \in E'_{nec}$, we choose an arbitrary edge in G that maps to e' and add it to the E_{nec} of G.

Analysis of `MSC`$^{\texttt{i}*}$. We prove that for DAG G' with new edge $\langle u', v' \rangle$, if there is a new SCC S' formed, the E'_{nec} identified by Algo. 5 is actually an optimal MSCS of S', which is achieved by leveraging the DAG property of G'. As E'_{nec} is an optimal MSCS in G', it indicates that the number of edges added into the updated E_{nec} 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_{nec} is minimum.

Algorithm 6: Incremental MSCSC: `MSC`$^{\texttt{i}}$

Input: Graph G with approximate MSCSC E_{nec}, 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_{nec} and a new G'

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

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

Theorem 4.2. *Given a graph G with 2-approximate MSCSC E_{nec}, after inserting an edge, suppose that the optimal solution before and after this update is E^*_{nec} and $E^{*'}_{nec}$, respectively, the number of edges added into the updated E_{nec} is minimum and equals to $|E^{*'}_{nec} - E^*_{nec}|$. And, the updated E_{nec} by Algo. 5 is 2-approximate.*

2-Approx Incremental MSCSC `MSC`$^{\texttt{i}}$. In the following, we present a more efficient 2-approximate solution `MSC`$^{\texttt{i}}$ in Algo. 6. `MSC`$^{\texttt{i}}$ 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 v' 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 low, dfn, and the stack $\mathcal{S}$ in previous methods. Algorithms 6 provides the pseudo code of `MSC`$^{\texttt{i}}$, which performs in a DFS manner starting from v'. Specifically, at Line 1 in Algo. 6, aff is initialized to store

Algorithm 7: Decremental MSCSC: $\texttt{MSC}^{\texttt{d}}$

Input: G, deleted edge $\langle u_d, v_d \rangle$, E_{tree}, $E_{lastdrop}$, E_{nec}
Output: Updated MSCSC E_{nec}
1 Delete this edge from G
2 **if** $\langle u_d, v_d \rangle \notin E_{nec}$ **then**
3   **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   Go to Line 11
8 **for each** *vertex* $u \in G_S$ **do**
9   **if** $visited[u] = false$ **then**
10         $ProcessNode(u)$
11 $E_{nec} \leftarrow E_{lastdrop} \cup \left(\bigcup_{\langle u,v \rangle \in E_{tree}, f(u)=f(v)} \langle u, v \rangle \right)$

Algorithm 8: SplitMSCS

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

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'_j is marked as visited. If the current node v'_j is u', which is the starting node of the new edge, then we add v'_j into aff and return true as the termination condition of recursion. A flag $\mathcal{R}$ indicating to merge or not is initialized as false at Line 14. For every outgoing edge $\langle v'_j, v'_k \rangle$ of v'_j in DAG G' (Line 15), if out-neighbor v'_k has been visited (Line 16, case 1) and is in aff (Line 17), but v'_j is not in aff yet, then we add v'_j into aff and add the edge into E'_{nec}. Case 1 is designed to facilitate Lemma 5 presented later. If v'_j is not visited, then procedure MergeMSCS is invoked for v'_k (Line 21 case 2), after which, $\mathcal{R}$ is set to true and v'_j is added into aff and E'_{nec} is updated accordingly (Lines 22-23).

Example 4.3. Fig. 5 shows an example of $\texttt{MSC}^{\texttt{i}}$*, with new edge in blue* $\langle v'_4, v'_1 \rangle$*. 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* $\langle v'_1, v'_2, v'_4 \rangle$*, 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'_4 *are added into* aff*. Then, we find nodes and locate necessary edges in other paths from* v'_1 *to* v'_4*. A path* $\langle v'_1, v'_3, v'_2 \rangle$ *is found where* v'_3 *is not in* aff*. Then we add* v'_3 *into* aff *and edges in this path become necessary. When we reach* $\langle v'_3, v'_4 \rangle$*, since* v'_3 *and* v'_4 *are both in the* aff*, then this edge is unnecessary. We can see that the nodes in* aff*,* $\{v'_1, v'_2, v'_3, v'_4\}$*, represent the MSCSs to merge, and the necessary edges in* G' *are* $E'_{nec} = \{\langle v'_1, v'_2 \rangle, \langle v'_2, v'_4 \rangle, \langle v'_1, v'_3 \rangle, \langle v'_3, v'_2 \rangle, \langle v'_4, v'_1 \rangle\}$*. 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 $\texttt{MSC}^{\texttt{i}}$**.** In Lemma 5, we prove that Algo. 6 finds a 2-approximate MSCS 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 $\texttt{MSC}^{\texttt{i}}$ are both $O(n' + m')$ as it 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* $\langle u', v' \rangle$*, 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 $\langle u_d, v_d \rangle$ 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 $\langle u_d, v_d \rangle$ is not in E_{nec}, then nothing needs to be done to maintain E_{nec}. If edge $\langle u_d, v_d \rangle$ 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 $\langle u_d, v_d \rangle$, then the MSCS does not need to split and we only need to update the relevant necessary edges in the path into 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 naïve 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

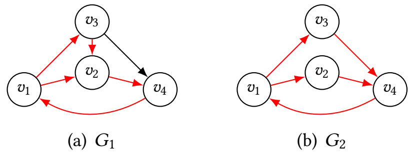


Figure 6: An example of decremental maintenance of E_{nec}.

is found in the updated G, indicating that this MSCS will not split. Thus, we should first determine whether the MSCS will split or not, and then locate new necessary edges to be updated in E_{nec}. Therefore, we present a decremental update method $\mathtt{MSC}^{\mathrm{d}}$ (Algo. 7) that starts DFS from u_d in G, and traverses every edge at most once. If the edge is not in E_{nec}, we can simply return at Line 3 after deleting the edge. Then we retrieve the induced subgraph G_S that is an SCC containing all nodes in the same MSCS as u_d at Line 4. The subsequent operations are operated on G_S. (Note that subgraph G_S is virtually induced in pseudo code for the ease of presentation. In our implementation, there is no need to actually extract G_S from G.) We set $depth$ to 1, mark all nodes unvisited, and initialize a redo flag to be false at Line 5. At Line 6 of Algo. 7, a procedure *SplitMSCS* (Algo. 8) 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 MSCSs by Algo. 4 for every node u to get them (Lines 8-10). Finally, E_{nec} is updated at Line 11.

As mentioned, procedure *SplitMSCS* (Algo. 8) determines if the MSCS splits or not (*i.e.*, if there is another path from u_d to v_d) 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_d to v_d, 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 Algo. 3. To tackle both scenarios simultaneously, a vital step in *SplitMSCS* different from Algo. 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 $\mathcal{S}$ 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 Algo. 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 to v_d is found at Line 1, to ensure 2-approximation, we verify if the number of edges in E_{nec} exceeds $2|V(G_S)| - 2$, the max possible number of necessary edges for 2-approximation in G_S, at Lines 2-3. If yes, we set the redo flag to be true, which will lead to the execution of Lines 8-10 in Algo. 7 to get new E_{nec}.

Example 4.5. Fig. 6 shows an example of decremental necessary edge maintenance. The red edges indicate the necessary edges in E_{nec}. Suppose that $\langle v_3, v_2\rangle$ in Fig. 6(a) is deleted. Since this edge is a necessary edge, we need to check whether there is an alternative path from v_3 to v_2 in the updated graph as shown in Fig. 6(b). A path $\langle v_3, v_4, v_1, v_2\rangle$ can be found, which indicates that this MSCS will not split. Then to maintain the connectivity of vertices in this path, edges $\{\langle v_3, v_4\rangle, \langle v_4, v_1\rangle, \langle v_1, v_2\rangle\}$ in this path are added into E_{nec} (Fig. 6(b)). Then, we terminate traversal without visiting $\langle v_1, v_3\rangle$ and $\langle v_2, v_4\rangle$. The final necessary edges are in red in Fig. 6(b).

Table 2: Statistics of Datasets. ($K = 10^3$, $M = 10^6$, $B = 10^9$)

Name	Dataset	V	E	$d = \frac{\lvert E\rvert}{\lvert V\rvert}$
EP	Epinions	75.9K	509K	6.7
YT	Youtube	1.14M	4.94M	4.3
IN	IN-2004	1.38M	16.5M	12
WF	Wikifr	3.33M	124M	37.1
EU	EU-2005	11.3M	380M	33.7
IT	IT-2004	41.3M	1.14B	27.5
T3W	TwitterWWW	41.7M	1.47B	35.3
FS	Friendster	68.3M	2.59B	37.8

Correctness and Complexity Analysis. If the deleted edge $\langle u_d, v_d\rangle$ is not in E_{nec}, then nothing needs to do. Otherwise, there are two cases. The first case is that the corresponding MSCS splits. In this case, $\mathtt{MSC}^{\mathrm{d}}$ performs like Algo. 3 inside the induced graph G_S. The second case is that the MSCS will not split, and we find another path from u_d to v_d and insert the edges on the path into E_{nec}, which maintains the connectivity from u_d to v_d. Thus, E_{nec} updated by Algo. 7 maintains the strong connectivity of G. The time and space complexities of Algo. 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 cost per edge. 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 Algo. 7), or the MSCS will not split (Lines 1-4 and 12-13 in Algo. 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_{nec}, after deleting an edge, the updated E_{nec} by Algo. 7 is 2-approximate.*

5 EXPERIMENTS

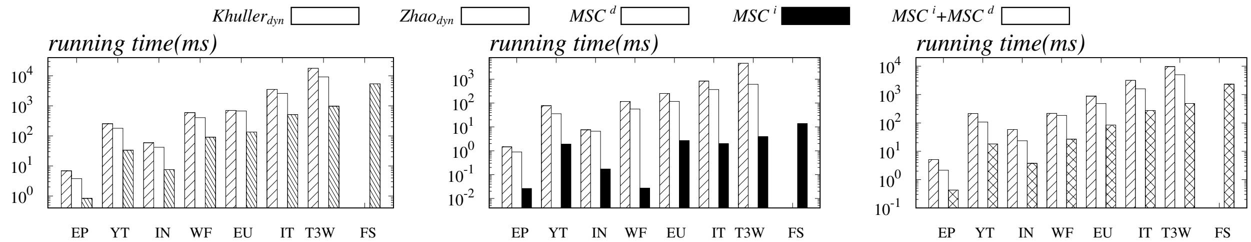
We conduct experiments on a Linux machine with an Intel Xeon 2.10GHz CPU and 504GB memory. All algorithms are in C++ and compiled via g++ with full optimization. Our code is at [1].

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. `Khuller` [25] runs in a near-linear time and is 7/4-approximate. Therefore, we extend `Khuller` and `Zhao` to MSCSC. For static graphs, `Khuller` and `Zhao` first apply Algo. 1 to detect SCCs and then detect MSCS of each SCC. For dynamic graphs, $\mathtt{Khuller}_{\mathtt{dyn}}$ and $\mathtt{Zhao}_{\mathtt{dyn}}$ 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 $R_{nec} = |E_{nec}|/|edges\ in\ SCCs|$, *i.e.*, the ratio of the number of edges in approximate E_{nec} over the number of all edges in SCCs of G. A lower necessary ratio R_{nec} indicates a tighter approximation.



(a) Average deletion time (Lower is better.) (b) Average insertion time (Lower is better.) (c) Average mixed update time (Lower is better.)

Figure 7: Dynamic MSCSC with edge deletion and insertion.

Table 3: Necessary edge ratio under update.

Dataset	R_{nec}				
	$\mathtt{MSC}^{\mathtt{d}}$	$\mathtt{MSC}^{\mathtt{i}}$	$\mathtt{Khuller}_{\mathtt{dyn}}$	$\mathtt{Zhao}_{\mathtt{dyn}}$	$\mathtt{MSC}^{\mathtt{i}}$ +$\mathtt{MSC}^{\mathtt{d}}$
EP	12.65%	12.60%	13.91%	12.03%	14.60%
YT	20.73%	20.71%	22.97%	19.84%	20.75%
IN	10.21%	10.21%	11.49%	10.07%	10.21%
WF	3.95%	3.95%	3.48%	2.81%	3.97%
EU	3.30%	3.30%	3.29%	2.92%	3.43%
IT	5.10%	5.10%	5.73%	5.00%	5.12%
T3W	4.07%	4.07%	5.17%	3.87%	5.53%
FS	5.42%	5.42%	OOT	OOT	5.07%

Table 4: Construction time and necessary edge ratio.

Dataset	CT (seconds)			R_{nec}		
	MSC	Khuller	Zhao	MSC	Khuller	Zhao
EP	0.014	0.0571	0.0386	13.30%	12.59%	11.95%
YT	0.313	1.39	0.852	22.55%	20.71%	19.82%
IN	0.236	1.43	0.735	11.47%	10.17%	10.07%
WF	3.27	12.9	8.72	3.42%	2.97%	2.81%
EU	6.02	29.5	18.3	3.30%	3.05%	2.92%
IT	12.8	88.6	40.7	5.73%	5.10%	5.00%
T3W	53.6	538	181	5.26%	4.07%	3.87%
FS	110	797	566	5.97%	5.07%	4.96%

5.2 MSCSC Evaluation

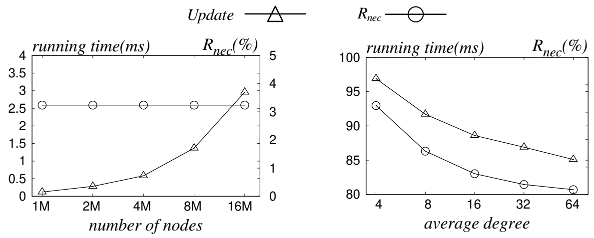
We evaluate the performance under three workloads: edge deletion, edge insertion and mixed workload. We also report the MSCSC construction performance and the scalability on synthetic graphs.

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_{nec}. Fig. 7(a) reports the average MSCSC maintenance time in milliseconds (*ms*) on all edge deletions over all datasets of our method $\mathtt{MSC}^{\mathtt{d}}$, $\mathtt{Khuller}_{\mathtt{dyn}}$, and $\mathtt{Zhao}_{\mathtt{dyn}}$. Observe that $\mathtt{MSC}^{\mathtt{d}}$ is consistently faster than $\mathtt{Zhao}_{\mathtt{dyn}}$ and $\mathtt{Khuller}_{\mathtt{dyn}}$, often by an order of magnitude. For instance, on T3W, a large graph with billions of edges, $\mathtt{MSC}^{\mathtt{d}}$ updates MSCSC in 960*ms* per edge deletion, which is 10 times faster than $\mathtt{Zhao}_{\mathtt{dyn}}$ that costs 9200*ms* and 20 times faster than $\mathtt{Khuller}_{\mathtt{dyn}}$ that takes 17800*ms*. Moreover, on the largest FS graph, $\mathtt{MSC}^{\mathtt{d}}$ is efficient, while $\mathtt{Khuller}_{\mathtt{dyn}}$ and $\mathtt{Zhao}_{\mathtt{dyn}}$ run OOT. The speedup of $\mathtt{MSC}^{\mathtt{d}}$ over the competitors validates the efficiency of the techniques proposed in Section 4.2 for dynamic MSCSC under edge deletion. $\mathtt{MSC}^{\mathtt{d}}$ only needs to focus on the local subgraph affected and scans the edges in the subgraph only once, while $\mathtt{Khuller}_{\mathtt{dyn}}$ and $\mathtt{Zhao}_{\mathtt{dyn}}$ need to compute from scratch and scan the subgraph twice. Further, in the second col umn of Tab. 3, after massive edge deletions, the necessary ratios of $\mathtt{MSC}^{\mathtt{d}}$ remain stable on all datasets with a negligible increase compared with $\mathtt{Khuller}_{\mathtt{dyn}}$ and $\mathtt{Zhao}_{\mathtt{dyn}}$, validating the effectiveness of our techniques for dynamic MSCSC under edge deletions, and indicating the better trade-off for efficiency achieved by $\mathtt{MSC}^{\mathtt{d}}$ 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 $\mathtt{MSC}^{\mathtt{i}}$ consistently outperforms $\mathtt{Zhao}_{\mathtt{dyn}}$ and $\mathtt{Khuller}_{\mathtt{dyn}}$ by a significant margin, often in orders of magnitude. On IT with 1.14 billion edges, $\mathtt{MSC}^{\mathtt{i}}$ runs in 2*ms* to maintain MSCSC per edge insertion, while $\mathtt{Zhao}_{\mathtt{dyn}}$ requires 370*ms*, which is 135 times slower and $\mathtt{Khuller}_{\mathtt{dyn}}$ requires 836*ms*, which is 418 times slower. On the largest FS graph with 68.3 million nodes and 2.59 billion edges, $\mathtt{Khuller}_{\mathtt{dyn}}$ and $\mathtt{Zhao}_{\mathtt{dyn}}$ run OOT. For edge insertions, $\mathtt{MSC}^{\mathtt{i}}$ works on the reduced DAG G' to identify the MSCSs that need to merge and then update E_{nec} in G accordingly, which explains its superiority compared with $\mathtt{Khuller}_{\mathtt{dyn}}$ and $\mathtt{Zhao}_{\mathtt{dyn}}$. Tab. 3 shows the necessary ratios R_{nec} of $\mathtt{MSC}^{\mathtt{i}}$ that is close to $\mathtt{Khuller}_{\mathtt{dyn}}$ and $\mathtt{Zhao}_{\mathtt{dyn}}$, 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 ($\mathtt{MSC}^{\mathtt{i}}$ +$\mathtt{MSC}^{\mathtt{d}}$), Fig. 7(c) shows the average update time and the last column in Tab. 3 reports R_{nec} under the mixed workload. In Fig. 7(c), $\mathtt{MSC}^{\mathtt{i}}$ +$\mathtt{MSC}^{\mathtt{d}}$ is 6X-7X faster than $\mathtt{Zhao}_{\mathtt{dyn}}$ in six datasets (EP, YT, IN, WF, EU, and IT), and one order of magnitude faster in T3W, and $\mathtt{Zhao}_{\mathtt{dyn}}$ runs OOT in FS, while $\mathtt{Khuller}_{\mathtt{dyn}}$ is even slower. Moreover, in Tab. 3, observe that R_{nec} 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_{nec} (*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_{nec}. Observe that R_{nec} of `MSC` is close to that of `Khuller` and `Zhao`, indicating that



(a) Vary graph size (b) Vary graph density

Figure 8: Scalability on Synthetic Graphs

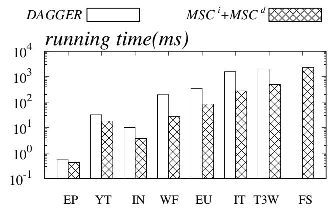


Figure 9: Fully Dynamic SCC Maintenance.

our method provides close practical approximation, which certifies the small theoretical guarantee gap among these methods.

Comparison on $\mathtt{MSC}^{\mathtt{i}*}$ in Algo. 5 and $\mathtt{MSC}^{\mathtt{i}}$ 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 $\mathtt{MSC}^{\mathtt{i}}$ yields higher efficiency with a significant speedup ratio over $\mathtt{MSC}^{\mathtt{i}*}$. The number of edges in E_{nec} of the optimal incremental solution $\mathtt{MSC}^{\mathtt{i}*}$ is always smaller than that of $\mathtt{MSC}^{\mathtt{i}}$, but they are close with small Δ values, indicating that $\mathtt{MSC}^{\mathtt{i}}$ in Algo. 6 is practically effective in maintaining tight E_{nec}, 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 $\mathtt{MSC}^{\mathtt{i}}$ +$\mathtt{MSC}^{\mathtt{d}}$ to handle a mixed workload with 10K edge insertions and 10K edge deletions, and report the average update time and necessary ratio R_{nec} 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_{nec} 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 $\mathtt{MSC}^{\mathtt{i}}$ +$\mathtt{MSC}^{\mathtt{d}}$ on the mixed workloads of these graphs and report running time and R_{nec} in Fig. 8(b). Observe that R_{nec} decreases as density increases. With higher graph density, the number of edges in E_{nec} remains relatively stable and is bounded by $2n-2$, but the number of edges in SCCs increases, resulting in the decrease of R_{nec}. Specifically, when the average degree varies from 4 to 64, the number of edges in E_{nec} is {1.18M,

Table 5: The update time of $\mathtt{MSC}^{\mathtt{i}}$ and $\mathtt{MSC}^{\mathtt{i}*}$ in ms, and the difference on the number of edges in their MSCSC answers.

Dataset	Time of $\mathtt{MSC}^{\mathtt{i}}$	Time of $\mathtt{MSC}^{\mathtt{i}*}$	**Speedup**	**Δ**
EP	0.0262	0.103	3.94	5
YT	1.9	15.3	8.1	3
IN	0.173	1.4	8.1	2
WF	0.0274	0.167	6.1	0
EU	2.68	19.1	7.1	4
IT	1.99	21.1	10.6	2
T3W	3.85	15.3	4	0
FS	13.8	79.4	5.8	0

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 E_{nec}, and nothing needs to be done. Similarly, in a denser graph, edge insertions may happen between nodes in the same MSCS, and E_{nec} 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 $\mathtt{Adam}_{\mathtt{SCC}}$ [24] theoretically achieves state-of-the-art worst-case time complexity $O(n^{1.529})$, which however is not practical with immense memory consumption. In experiments, $\mathtt{Adam}_{\mathtt{SCC}}$ runs out of memory (OOM) even for the smallest data EP. To achieve the time complexity, $\mathtt{Adam}_{\mathtt{SCC}}$ 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 $\mathtt{DAGGER}$ [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 ($\mathtt{MSC}^{\mathtt{i}}$ +$\mathtt{MSC}^{\mathtt{d}}$) 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 by replacing SCCs with MSCSC. Note that our MSCSC is capable for efficient dynamic SCC-based index maintenance, such as TOL [56] and $\mathtt{DAGGER}$ [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

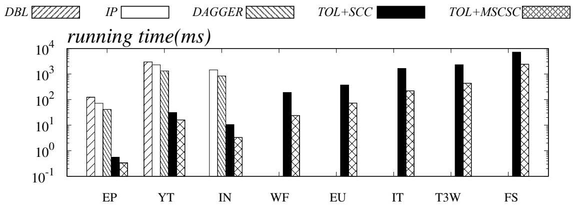


Figure 10: Reachability Index Maintenance Efficiency.

MSCSs 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_{nec} 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.

6 RELATED WORK

In addition to existing studies on MSCS and SCC reviewed in Section 2, we review the related work on reachability queries here, and a survey is in [54]. There exists a plethora of reachability query methods on directed graphs [3, 9–13, 13, 18, 20, 22, 23, 36, 40, 41, 44, 46, 47, 49–52, 52, 53, 56], which can be divided into three categories: (i) index-free methods [15, 36, 40] that directly conduct online breath-first, depth-first, or random walk traversals on graphs for reachability query processing; (ii) index-only solutions [11, 13, 19, 46, 47, 49, 51, 56], which build efficient indices and reachability query processing is all handled with the index only; (iii) index+traversal methods [31, 41, 44, 50, 52, 52, 53], which also build indices, but leverage both the indices and graphs to process reachability queries.

Table 6: Reachability Query Time in nanoseconds.

Dataset	DBL	IP	DAGGER	TOL+SCC	TOL+MSCSC
EP	25	188	3.23M	61	61
YT	132	117	78.1M	122	122
IN	26.2K	201	16.9M	114	114
WF	100	189	9.25M	180	180
EU	602	197	609M	241	241
IT	8K	119	483M	152	152
T3W	2.38K	121	817M	169	169
FS	5.96K	210	2.64B	241	241

Depending on whether transforming the original graph into a DAG or not, reachability indexes can be divided into SCC-based indexes [16, 21, 34, 41, 44, 47, 50, 52, 53, 56] and non-SCC indexes [13, 21, 33, 39, 46, 51]. A main methodology for SCC-based indexes is to first transform the input G into a DAG G', which is a reduced graph by shrinking each SCC of G into a single node in G', and then perform reachability query processing with the assistance of G'. The reduced graph is typically one to two orders of magnitude smaller than the original input graph, which can help significantly reduce the online traversal costs and reduce the index size and construction cost. Dynamic SCC-based index methods include `DAGGER` [53], TOL [56] and `IP` [50]. For example, DAGGER extended from GRAIL [52] is a dynamic method with an interval labeling index and SCC maintenance to handle reachability queries on dynamic graphs. TOL [56] adopts 2-hop indexing techniques over a reduced DAG graph, and supports node insertion and node deletion. `IP` [50] explores the randomness to answer reachability queries. It needs to be mentioned that TOL and `IP` assume that there are no SCC merges or splits. For non-SCC indexes, `DBL` [33] is a recent method on dynamic graphs. It builds on two complementary indexes: Dynamic Landmark (DL) label and Bidirectional Leaf (BL) label. In our use case study, we extend TOL to handle edge updates including insertions and deletions, and our methods are readily applicable to extend TOL and boost its index update efficiency on dynamic graphs with a mixed workload of edge insertions and deletions.

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 $\mathtt{MSC}^{\mathtt{i}}$ and $\mathtt{MSC}^{\mathtt{d}}$ 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.

REFERENCES

[1] 2023. Our Implementation and Technical Report. https://github.com/jerchenxin/mscsc.

[2] Amir Abboud and Virginia Vassilevska Williams. 2014. Popular Conjectures Imply Strong Lower Bounds for Dynamic Problems. In *FOCS*. 434–443.

[3] Rakesh Agrawal, Alexander Borgida, and H. V. Jagadish. 1989. Efficient Management of Transitive Relationships in Large Data and Knowledge Bases. In *SIGMOD*. 253–262.

[4] Yash P. Aneja, Ramaswamy Chandrasekaran, Xiangyong Li, and K. P. K. Nair. 2010. A branch-and-cut algorithm for the strong minimum energy topology in wireless sensor networks. *Eur. J. Oper. Res.* 204, 3 (2010), 604–612.

[5] Michael A Bender, Jeremy T Fineman, Seth Gilbert, and Robert E Tarjan. 2015. A new approach to incremental cycle detection and related problems. *TALG* 12, 2 (2015), 1–22.

[6] Aaron Bernstein, Aditi Dudeja, and Seth Pettie. 2021. Incremental SCC Maintenance in Sparse Graphs. In *ESA*. 14:1–14:16.

[7] Aaron Bernstein, Maximilian Probst, and Christian Wulff-Nilsen. 2019. Decremental strongly-connected components and single-source reachability in near-linear time. In *STOC*. 365–376.

[8] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *WWW*. 595–601.

[9] Ramadhana Bramandia, Byron Choi, and Wee Keong Ng. 2010. Incremental Maintenance of 2-Hop Labeling of Large Graphs. *TKDE* 22, 5 (2010), 682–698.

[10] Yangjun Chen and Yibin Chen. 2008. An Efficient Algorithm for Answering Graph Reachability Queries. In *ICDE*. 893–902.

[11] James Cheng, Silu Huang, Huanhuan Wu, and Ada Wai-Chee Fu. 2013. TF-Label: a topological-folding labeling scheme for reachability querying in a large graph. In *SIGMOD*. 193–204.

[12] Jiefeng Cheng, Jeffrey Xu Yu, Xuemin Lin, Haixun Wang, and Philip S. Yu. 2006. Fast Computation of Reachability Labeling for Large Graphs. In *EDBT*, Vol. 3896. 961–979.

[13] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2003. Reachability and Distance Queries via 2-Hop Labels. *SIAM J. Comput.* 32, 5 (2003), 1338–1355.

[14] Edsger Wybe Dijkstra. 1976. *A discipline of programming.*

[15] Igor Gorodezky and Igor Pak. 2014. Generalized loop-erased random walks and approximate reachability. *Random Struct. Algorithms* 44, 2 (2014), 201–223.

[16] Kathrin Hanauer, Christian Schulz, and Jonathan Trummer. 2022. O'Reach: Even Faster Reachability in Large Graphs. *ACM J. Exp. Algorithmics* 27 (2022), 4.2:1–4.2:27.

[17] Sungpack Hong, Nicole C. Rodia, and Kunle Olukotun. 2013. On fast parallel detection of strongly connected components (SCC) in small-world graphs. In *SC*. 92:1–92:11.

[18] H. V. Jagadish. 1990. A Compression Technique to Materialize Transitive Closure. *TODS* 15, 4 (1990), 558–598.

[19] Ruoming Jin, Hui Hong, Haixun Wang, Ning Ruan, and Yang Xiang. 2010. Computing label-constraint reachability in graph databases. In *SIGMOD*. 123–134.

[20] Ruoming Jin, Ning Ruan, Yang Xiang, and Haixun Wang. 2011. Path-tree: An efficient reachability indexing scheme for large directed graphs. *TODS* 36, 1 (2011), 7:1–7:44.

[21] Ruoming Jin and Guan Wang. 2013. Simple, Fast, and Scalable Reachability Oracle. *Proc. VLDB Endow.* 6, 14 (2013), 1978–1989.

[22] Ruoming Jin, Yang Xiang, Ning Ruan, and David Fuhry. 2009. 3-HOP: a high-compression indexing scheme for reachability query. In *SIGMOD*. 813–826.

[23] Ruoming Jin, Yang Xiang, Ning Ruan, and Haixun Wang. 2008. Efficiently answering reachability queries on very large directed graphs. In *SIGMOD*. 595–608.

[24] Adam Karczmarz and Marcin Smulewicz. 2023. On Fully Dynamic Strongly Connected Components. In *ESA (LIPIcs)*, Vol. 274. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 68:1–68:15.

[25] Samir Khuller, Balaji Raghavachari, and Neal E. Young. 1995. Approximating the Minimum Equivalent Digraph. *SICOMP* 24, 4 (1995), 859–872.

[26] Samir Khuller, Balaji Raghavachari, and Neal E. Young. 1996. On Strongly Connected Digraphs with Bounded Cycle Length. *DAM* 69, 3 (1996), 281–289.

[27] Jérôme Kunegis. 2013. KONECT: the Koblenz network collection. In *WWW*. 1343–1350.

[28] Jakub Łącki. 2013. Improved deterministic algorithms for decremental reachability and strongly connected components. *TALG* 9, 3 (2013), 1–15.

[29] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[30] Guohui Li, Zhe Zhu, Cong Zhang, and Fumin Yang. 2014. Efficient decomposition of strongly connected components on GPUs. *JSA* 60, 1 (2014), 1–10.

[31] Lei Li, Wen Hua, and Xiaofang Zhou. 2017. HD-GDD: high dimensional graph dominance drawing approach for reachability query. *WWW* 20, 4 (2017), 677–696.

[32] Wenjie Li, Lei Zou, Peng Peng, and Zheng Qin. 2021. NREngine: A Graph-Based Query Engine for Network Reachability. In *Database Systems for Advanced Applications. DASFAA 2021 International Workshops: BDQM, GDMA, MLDLDSA, MobiSocial, and MUST, Taipei, Taiwan, April 11–14, 2021, Proceedings 26*. Springer, 90–106.

[33] Qiuyi Lyu, Yuchen Li, Bingsheng He, and Bin Gong. 2021. DBL: Efficient Reachability Queries on Dynamic Graphs. In *DASFAA (Lecture Notes in Computer Science)*, Vol. 12682. Springer, 761–777.

[34] Florian Merz and Peter Sanders. 2014. PReaCH: A Fast Lightweight Reachability Index Using Pruning and Contraction Hierarchies. In *Algorithms - ESA 2014 - 22th Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings (Lecture Notes in Computer Science)*, Andreas S. Schulz and Dorothea Wagner (Eds.), Vol. 8737. Springer, 701–712.

[35] Mark Newman. 2018. *Networks*. Oxford university press.

[36] Yue Pang, Lei Zou, and Yu Liu. 2023. IFCA: Index-Free Community-Aware Reachability Processing Over Large Dynamic Graphs. In *ICDE*. IEEE, 2220–2234.

[37] Liam Roditty and Uri Zwick. 2008. Improved Dynamic Reachability Algorithms for Directed Graphs. *SICOMP* 37, 5 (2008), 1455–1471.

[38] Mohamed Sarwat, Sameh Elnikety, Yuxiong He, and Mohamed F Mokbel. 2013. Horton+ a distributed system for processing declarative reachability queries over partitioned graphs. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1918–1929.

[39] Ralf Schenkel, Anja Theobald, and Gerhard Weikum. 2005. Efficient Creation and Incremental Maintenance of the HOPI Index for Complex XML Document Collections. In *ICDE*. 360–371.

[40] Neha Sengupta, Amitabha Bagchi, Maya Ramanath, and Srikanta Bedathur. 2019. ARROW: Approximating Reachability Using Random Walks Over Web-Scale Graphs. In *ICDE*. 470–481.

[41] Stephan Seufert, Avishek Anand, Srikanta J. Bedathur, and Gerhard Weikum. 2013. FERRARI: Flexible and efficient reachability range assignment for graph indexing. In *ICDE*. IEEE Computer Society, 1009–1020.

[42] Micha Sharir. 1981. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications* 7, 1 (1981), 67–72.

[43] George M. Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. 2014. BFS and Coloring-Based Parallel Algorithms for Strongly Connected Components and Related Problems. In *IPDPS*. 550–559.

[44] Jiao Su, Qing Zhu, Hao Wei, and Jeffrey Xu Yu. 2017. Reachability Querying: Can It Be Even Faster? *TKDE* 29, 3 (2017), 683–697.

[45] Robert Endre Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SICOMP* 1, 2 (1972), 146–160.

[46] Silke Trißl and Ulf Leser. 2007. Fast and practical indexing and querying of very large graphs. In *SIGMOD*. 845–856.

[47] Renê Rodrigues Veloso, Loïc Cerf, Wagner Meira Jr., and Mohammed J. Zaki. 2014. Reachability Queries in Very Large Graphs: A Fast Refined Online Search Approach. In *EDBT*. OpenProceedings.org, 511–522.

[48] Adrian Vetta. 2001. Approximating the minimum strongly connected subgraph via a matching lower bound. In *SODA*. 417–426.

[49] Haixun Wang, Hao He, Jun Yang, Philip S. Yu, and Jeffrey Xu Yu. 2006. Dual Labeling: Answering Graph Reachability Queries in Constant Time. In *ICDE*. 75.

[50] Hao Wei, Jeffrey Xu Yu, Can Lu, and Ruoming Jin. 2018. Reachability querying: an independent permutation labeling approach. *VLDB J.* 27, 1 (2018), 1–26.

[51] Yosuke Yano, Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths. In *CIKM*. 1601–1606.

[52] Hilmi Yildirim, Vineet Chaoji, and Mohammed Javeed Zaki. 2010. GRAIL: Scalable Reachability Index for Large Graphs. *Proc. VLDB Endow.* 3, 1 (2010), 276–284.

[53] Hilmi Yildirim, Vineet Chaoji, and Mohammed J. Zaki. 2013. DAGGER: A Scalable Index for Reachability Queries in Large Dynamic Graphs. *CoRR* abs/1301.0977 (2013).

[54] Chao Zhang, Angela Bonifati, and M. Tamer Özsu. 2023. An Overview of Reachability Indexes on Graphs. In *SIGMOD Conference Companion*. ACM, 61–68.

[55] Liang Zhao, Hiroshi Nagamochi, and Toshihide Ibaraki. 2003. A linear time 53-approximation for the minimum strongly-connected spanning subgraph problem. *Information processing letters* 86, 2 (2003), 63–70.

[56] Andy Diwen Zhu, Wenqing Lin, Sibo Wang, and Xiaokui Xiao. 2014. Reachability queries on large dynamic graphs: a total order approach. In *SIGMOD*. 1323–1334.