Detecting Strongly Connected Components for Scholarly Data

Junfeng Liu, Shuai Ma and Hanqing Chen
State Key Laboratory of Software Development Environment, Beihang University, Beijing, China
{liujunfeng, mashuai, chenhanqing}@buaa.edu.cn

***Abstract*—Strongly connected component (SCC) detection is a fundamental graph analytic algorithm on citation graphs that plays a significant role in scholarly data analysis tasks. However, all the existing SCC detection methods are designed for general graphs. In this study, we focus on detecting strongly connected components (SCCs) of citation graphs for scholarly data. (1) We first develop an efficient static SCC detection algorithm by dividing edges into three types and exploiting the properties of different types of edges to reduce the traversal of unnecessary nodes and edges. (2) After analyzing the key design issues for incremental SCC detection, we design an efficient bounded incremental algorithm to handle continuous single updates by dynamically maintaining the citation graph partition and local topological order. (3) We then design an efficient bounded batch incremental algorithm by reducing the traversal of unnecessary edges, based on our single update algorithm, to further improve the efficiency for continuous batch updates. (4) Finally, we conduct an experimental study to verify the efficiency and impact of our static and incremental SCC detection algorithms for citation graphs.**

***Index Terms*—incremental strongly connected component detection, scholarly data analysis, graph partition, topological order**

I. Introduction

In recent years, the rapid development of science and technology comes with the prosperity of scientific publications, and scholarly data has drawn significant attention from both industrial and academic communities. Analysis of scholarly data helps to understand evolving scientific trends and assists scholars to find needed information timely and conveniently [1–3], among which citation graphs analyses play an important role, such as scholarly article ranking [4–6], scholarly community detection [7–9], scholar name disambiguation [10–12], scholarly data visualization [13–16] and scholarly search and analytic systems [1–3, 17].

Strongly connected components (SCCs) exist commonly in citation graphs, which are typically caused by two reasons. (1) There is a time gap between the accepted and published time of articles, and multiple articles cite each other. (2) Scholarly systems may wrongly treat the different articles with similar or same titles as one article. The real-life SCC examples are illustrated in Example 1.

Example 1: (1) Fig.1 demonstrates an SCC with two nodes formed by the *time gap* in the citation graph of DBLP [18], where the nodes A and B represent two articles published at OSDI 2014. After the two articles are accepted, the committee finds they study the same topic, and the authors are further asked to compare their systems with each other, which leads to the mutual citations and formation of an SCC.

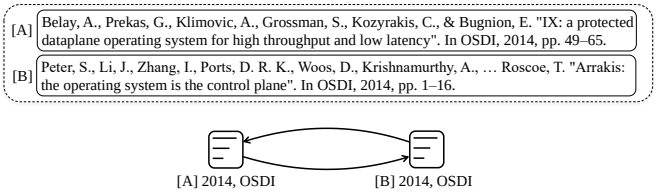
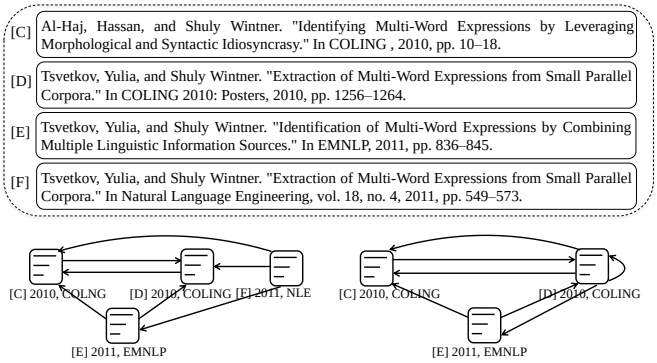


Figure 1. A real-life SCC formed by the time gap



(b) Citations in articles (c) Citations in DBLP [18]

Figure 2. A real-life SCC formed by the two reasons

(2) Fig. 2 depicts another real-life SCC of DBLP formed due to the two reasons. The nodes C, D, E and F are the articles published at COLING 2010, COLING 2010, EMNLP 2011 and NLE 2011, respectively, and the citations in articles are shown in Fig. 2(b). However, in the citations of DBLP, there forms an SCC with nodes C, D and E, as shown in Fig. 2(c). This is because (i) the edges (C, D) and (D, C) are formed by the *time gap*, similar to Fig. 1. (ii) The edges (D, C), (D, E) and (D, D) are caused by *wrongly* treating F as D, where article F has the same title and authors as article D. □

Detecting strongly connected component (SCC) is a fundamental graph analytic problem on citation graphs that have significant impacts on the computational efficiency and data quality of citation graph based tasks. (1) Scholarly article ranking commonly employs PageRank [5, 6, 19], where SCCs are detected first and each SCC is treated as a block in the process [6, 19]. (2) The SCCs of citation graphs are detected as a strongly cohesive pattern in the core-based community detection task [8]. (3) The detected SCCs provide views and perspectives for visualizing the citation graph topology [13], where the incremental visual exploration is

supported by incremental SCC detection [16, 20]. (4) SCC detection is the first step to identify SCCs caused by incorrect citations, and improve the quality of the scholarly data analysis tasks [21, 22].

There exist quite a few studies designed for static and incremental SCC detection of general graphs. For instance, [23–27] study static SCC detection, and traverse the entire graphs using depth-first search (DFS); [28–30] study incremental SCC detection by incrementally maintaining the (weak) topological order of the nodes in the entire graphs. However, different from general graphs, edges in citation graphs typically follow an order of the published time of articles. Applying these methods directly to citation graphs leads to traverse unnecessary nodes and edges, which is too costly. That is, the properties of scholarly data should be exploited for the SCC detection of citation graphs.

To our knowledge, we are the first study on detecting SCCs tailored to citation graphs. However, the efficient SCC detection on citation graphs needs to deal with two issues. (1) How to employ the properties of citation graphs to detect SCCs, as all the existing SCC detection methods are for general graphs [23–37]. (2) How to incrementally detect SCCs, as citation graphs are typically large and continuously growing.

Contributions. To this end, we propose efficient static and bounded incremental methods to detect strongly connected components in citation graphs.

(1) We develop an efficient static algorithm staDSCC to detect SCCs in citation graphs (Section III), by dividing edges into three types and exploiting the properties of different types of edges to reduce the traversal of unnecessary nodes and edges.

(2) We develop an efficient bounded incremental algorithm sinDSCC to handle continuous single updates by dynamically maintaining the citation graph partition and local topological order (Section V), based on the analyses of incremental SCC detection (Section IV).

(3) We design an efficient bounded batch incremental algorithm batDSCC by reducing the traversal of unnecessary edges (Section VI), based on single update algorithm sinDSCC, to further improve the efficiency for continuous batch updates.

(4) Finally, we have conducted an extensive experimental study on four real-life citation graphs: AAN [38], DBLP [18], ACM [18] and MAG [17] (Section VII). We find that (a) our static algorithm staDSCC is both time and space efficient. Indeed, our staDSCC is $(4.9, 5.9, 6.0, 6.7)$ times faster and uses $(2.0, 4.2, 3.2, 5.4)$ times less space than (Pearce2 [25], Tarjan [27], Gabow [24] and Kosaraju [26]) on average, respectively. (b) Our batch incremental algorithm batDSCC is on average $(5.8, 5.0, 35.4+, 51.6+, 5.0)$ times faster than (AHRSZ [31], HKMST [30], incSCC$^+$ [29], MNR [35] and PK$_2$ [37]), respectively. (c) The SCCs have significant impacts on the PageRank scores, and the score gaps are on average (2.9%, 1.1%, 1.3%, 1.1%) on (AAN, DBLP, ACM, MAG) after removing SCCs, respectively.

II. Preliminary

In this section, we introduce basic concepts.

Citation graph. A *citation graph* $G(V, E)$ is a directed graph, where (1) V is a finite set of nodes such that each node represents an article associated with its published time (here we use the *year* as the default time granularity along the same setting with existing scholarly databases), and (2) $E \subseteq V \times V$ is a finite set of edges, in which an edge $(u, v) \in E$ denotes a directed edge from node u to v representing article u cites article v, and u and v are also called the $tail$ and $head$ of edge (u, v), respectively.

Strongly connected component (SCC). An SCC in a directed graph G is a maximal subgraph where there exists a path from each node to all the other ones. We focus on the detection of non-singleton SCCs having more than one node in this study.

We next introduce three concepts mainly related to the incremental SCC detection.

(Weak) topological order [28, 30]. It is known that the strongly connected components can be ordered in a topological manner [23], and existing incremental algorithms are typically designed based on maintaining a (weak) topological order [28–30]. A *topological order* for a directed acyclic graph (DAG) is a *total order* "$\prec$" of its nodes. Generally, topological orders are not unique, and a topological order is valid if nodes $u \prec v$ for all edges (u, v), and is invalid, otherwise. A *weak topological order* for a DAG is a *partial order* of the nodes such that nodes $u \prec v$ if (u, v) is an edge (that is, unreachable nodes may share the same order [28]).

Ordered list [39, 40]. Although one can maintain the topological order of a DAG by mapping each node v to a unique integer $ord(v)$ in $\{1, ..., |V|\}$, it is not flexible for the incremental order maintenance, *e.g.,* inserting new orders. Ordered lists are proposed to solve this issue such that the order $ord(v)$ of node v may be larger than $|V|$. Following the definition of the topological order, an ordered list maps each node to a unique order ord such that for each edge (u, v), $ord(u) \prec ord(v)$, and maintains the orders. It supports four operations within $O(1)$ time by employing a two-level list to shave the log factor of weight-balanced trees [39, 40]. (1) insertAfter(u, v): insert item v immediately after item u in the topological order. (2) insertBefore(u, v): insert item v immediately before item u in the order. (3) order(u, v): determine whether item u precedes v in the order or not. (4) delete(u): delete item u from the order. Note that, when it performs insertAfter (or insertBefore)(u, v), only a new *order* is created after (or before) item u, and the topological orders of all nodes except v remain unchanged.

Cover of affected node pairs of edge insertion. Although one can easily traverse the entire graph using DFS to incrementally detect SCCs, it is time-consuming as too many nodes need to be visited. We define the minimum cover of affected node pairs of edge insertion to capture the least amount of nodes to be visited. For an edge insertion (u, v) on graph $G(V, E)$, its affected node pairs ANP are those pairs (s, t) with s, t in V and $s \rightsquigarrow t \vee ord(s) \succ ord(t) \vee s = t$, where $s \rightsquigarrow t$ denotes there is a path from s to t in G. A cover K of the ANP is a set of nodes such that for any node pair $(s, t) \in$ ANP, $s \in K$

Table I
MAIN NOTATIONS

Notations	Descriptions
$G(V, E)$	Citation graph G with node set V and edge set E
$\Delta G(V_\Delta, E_\Delta)$	Increments to graph G (node and edge insertions)
E_{n2o}	Edge set of newly published articles cite the old ones
E_{s2s}	Edge set of all same-year citations
E_{s2si}	Edge set of same-year citations in year i
E_{o2n}	Edge set of old published articles cite the newly ones
$G_m(V_m, E_m)$	Subgraph of nodes and edges reachable from the edge heads in E_{o2n}
$G_{si}(V_{si}, E_{si})$	Subgraph of nodes and edges in year i and not in G_m reachable from the edge heads in E_{s2si}
$G_s(V_s, E_s)$	Set of subgraphs G_{si}
$G_r(V_r, E_r)$	Subgraph of nodes $V \setminus V_m \setminus V_s$ and their connected edges
E_c	Set of cross edges $(E \setminus E_m \setminus E_s \setminus E_r)$
V_{ns}	Reachable nodes from the edge heads of E_{o2n} and E_{s2s}
E_{ns}	Reachable edges from the edge heads of E_{o2n} and E_{s2s}

or $t \in K$. That is, a cover K contains those nodes that violate the topological order. Observe that the entire set V of nodes is a cover of the ANP of the edge insertion (u, v).

A cover K_{min} of the affected node pairs is minimum iff $\|K_{min}\| \leq \|K\|$ for all the other covers K. Here $\|K\|$ is the extended size of a cover K, which is the sum of the *extended-out size* of $\|K_f\|$ and the *extended-in size* of $\|K_b\|$ such that $\|K_f\| = |K_f| + |N^+(K_f)|$, where $K_f = \{s \in K | v \rightsquigarrow s\}$, v is the head of edge insertion (u, v) and $N^+(K_f)$ is the set of out-neighbors of K_f. $\|K_b\| = |K_b| + |N^-(K_b)|$, where $K_b = \{s \in K | s \rightsquigarrow u\}$, u is the tail of edge insertion (u, v) and $N^-(K_b)$ is the set of in-neighbors of K_b.

In fact, K_{min} measures the least amount of nodes to be visited when detecting SCCs and maintaining topological orders, where the out or in-neighbors are required to be traversed [36]. This is different from [31, 36] assuming that edge insertion (x, y) does not introduce any SCCs, whose cover definition is to capture the nodes with invalid topological orders of a DAG only, and is only used to maintain topological orders, and does not fit for detecting SCCs.

The main notations are summarized in Table I.

III. STATIC SCC DETECTION

Citation graphs are typically large and continuously growing, hence incremental algorithms are essentially needed, which is our main focus of this study. However, the incremental algorithms are commonly designed based on the results of static algorithms. In this section, we hence first analyze the properties of citation graphs, and present a static algorithm (an improved version of Pearce2 [25], by employing the properties of citation graphs) to facilitate the incremental detection of SCCs in citation graphs.

A. Analyses of Citation Graphs

We first explore the properties of citation graphs, which guide the design of our SCC detection algorithms.

To do this, we divide the edges in E into three disjoint types, using $year$ as the default time granularity:

(1) E_{n2o} represents the set of edges (u, v) that the $u.year > v.year$, *i.e.,* a newly published article u cites an old published one v.

(2) E_{s2s} represents the set of edges (u, v) that the $u.year = v.year$, *i.e.,* an article u cites another one v published in the same year. We also use E_{s2si} to denote the set of edges in E_{s2s} published in year i.

(3) E_{o2n} represents the set of edges (u, v) that the $u.year < v.year$, *i.e.,* an old published article u cites a newly published one v.

Note that it is obvious that E_{n2o}, E_{s2s} and E_{o2n} are mutually disjoint and $E = E_{n2o} \cup E_{s2s} \cup E_{o2n}$.

Property 1: *For any non-singleton* SCCs *in a citation graph, there must exist an edge e such that $e \in E_{o2n}$ or E_{s2s} and the in-degree of its tail is not* 0. □

Assume that there exists a non-singleton subgraph is an SCC such that all its edges belong to E_{n2o}. We can map each node v of the subgraph to an integer $ord(v)$ such that $ord(v)$ equals $-v.year$. For each edge $(u, v) \in E_{n2o}$ of the subgraph, we have $ord(u) \prec odr(v)$ as $u.year > v.year$. That is the order ord of nodes of this subgraph essentially form a topological order. Hence the subgraph is a DAG, which is a contradiction. From this, the property holds. The property reveals that one only needs to traverse the edges in E_{o2n} and E_{s2s} to detect all the SCCs. Further, there is no need to traverse the edges whose in-degrees of their tails are 0, as they cannot form any SCCs.

Property 2: *Statistical analysis reveals that the distribution of three types of edges E_{o2n}, E_{s2s} and E_{n2o} is seriously unbalanced,* i.e., *the sizes of E_{o2n} and E_{s2s} are much less than the one of E_{n2o}.* □

We perform a statistical analysis on four real-life citation graphs (AAN [38], DBLP [18], ACM [18] and MAG [17]), as illustrated in Table II. On citation graphs AAN, ACM and MAG, E_{n2o} occupies more than 95%, while E_{s2s} and E_{o2n} only account for less than 4% and 0.8%, respectively, and on DBLP, E_{n2o} of DBLP occupies 77%, while E_{s2s} and E_{o2n} only account for 21.3% and 1.68%, respectively.

From these two properties, it is easy to see that SCCs can be detected by traversing edges in E_{o2n} and E_{s2s} only, which potentially improves the efficiency by avoiding unnecessary node and edge traversals.

B. Algorithm for Static SCC Detection

We next present our static SCC detection algorithm based on the properties of citation graphs.

The main result is stated below.

Theorem 1: *Given a citation graph $G(V, E)$, there exists an algorithm that correctly detects all* SCCs *in $\Theta(|V_{ns}| + |E_{ns}|)$ time, where V_{ns} and E_{ns} are the nodes and edges that are reachable from the edge heads of E_{o2n} and E_{s2s}, respectively.* □

Pearce2 [25] is an up-to-date static strongly connected component detection algorithm running in $\Theta(|V| + |E|)$ time.

Table II
STATISTICS OF REAL-LIFE CITATION GRAPHS

Datasets	$\|V\|$	$\|E\|$	$\frac{\|E_{n2o}\|}{\|E\|}$	$\frac{\|E_{s2s}\|}{\|E\|}$	$\frac{\|E_{o2n}\|}{\|E\|}$	|SCCs|	$\frac{\|\text{SCCs=1}\|}{\|\text{SCCs}\|}$	$\frac{\|\text{SCCs=2}\|}{\|\text{SCCs}\|}$	$\frac{\|\text{SCCs>10}\|}{\|\text{SCCs}\|}$	Largest SCC
AAN	18K	83K	95.92%	4.03%	0.05%	17K	98.63%	1.13%	0.0170%	20
DBLP	3,140K	6,150K	77.00%	21.33%	1.68%	3,135K	99.86%	0.12%	0.0004%	23
ACM	2,382K	8,639K	96.27%	2.95%	0.77%	2,373K	99.72%	0.23%	0.0012%	40
MAG	183,686K	730,799K	97.71%	2.11%	0.19%	183,451K	99.91%	0.16%	0.0004%	126

AAN [38], DBLP [18], ACM [18] and MAG [17] are four real-life citation graphs.

Input: Citation graph $G(V, E)$, E_{o2n}, E_{s2s}, max_year, min_year.
Output: Set of SCCs mSCCs, inGm, inGsi and $rindex$.

1. $rindex[v] = 0$, inGm$[v] = false$, inGsi$[v] = 0$ for $\forall v \in V$;
2. **for each** edge $e \in E_{o2n}$
3. **if** $rindex[e.head] = 0$ **then** visit1$(e.head)$;
4. **for each** i from max_year **to** min_year **do**
5. **for each** edge $e \in E_{s2si}$
6. **if** $rindex[e.head] = 0$ **then** visit2$(e.head, i)$;
7. Generate mSCCs by gathering from $rindex$;
8. **return** mSCCs, inGm, inGsi and $rindex$.

Procedure visit1(v)

1. $root = true$; inGm$[v] = true$; /*root is a local variable*/
2. $rindex[v] = visitid$; $visitid = visitid + 1$;
3. **for each** $(v, w) \in E$ **do**
4. **if** $rindex[w] = 0$ **then** visit1(w);
5. **if** $rindex[w] < rindex[v]$ **then**
6. $rindex[v] = rindex[w]$; $root = false$;
7. **if** $root$ **then**
8. Pop out SCC members from S;
9. Assign SCC members with the same component id c;
10. Update c and $visitid$;
11. **else** $S.push(v)$. /*S is a stack*/

Figure 3. Algorithm staDSCC

It is designed based on Tarjan [27] to reduce the space requirement, which combines the two needed arrays of Tarjan into one array and has shown the ability to handle larger graphs in practice [25]. Specifically, Pearce2 detects the SCCs via DFS to generate a spanning forest, and each SCC is a subtree of the spanning forest. It identifies an SCC by finding the root of the subtree (*i.e., component root*) in the spanning forest and assigns a unique identifier to each component of the graph.

We next present the details of algorithm staDSCC, shown in Fig. 3, to detect SCCs by traversing from the edge heads of E_{o2n} and E_{s2s}, which is inspired by Pearce2 and uses Properties 1 & 2 in Section III-A.

Data structures. Our static algorithm staDSCC involves 4 data structures, *i.e.,* mSCCs, inGm, inGsi and $rindex$. (1) Array mSCCs is the set of SCCs storing the members of each SCC. (2) Array inGm is a flag map indicating whether a node is visited by the heads of edges in E_{o2n}. (3) Array inGsi maps a node to its associated $year$ if it is visited by the heads of edges in E_{s2s}. (4) Array $rindex$ is utilized to find the component root, and maps each visited node to the visit index of its *local root* during DFS, where the local root of a node v is the visited node reachable from v with the smallest visit index. After DFS is finished, $rindex$ maps each visited node to its component identifier.

Procedure visit1 detects the SCCs involving within E_{o2n}. Given an unvisited head v in E_{o2n}, it traverses G using DFS to find the SCCs, where the component root of SCC is first found when $rindex$ has not changed after visiting any successors. (1) Local variable $root$ for identifying the component root of an SCC, and inGm are set to $true$, $rindex[v]$ is initialized to the visit index $visitid$, and $visitid$ is increased by 1 (lines 1, 2). (2) For each unvisited successor w of v, it recursively calls visit1 (lines 3, 4). For each visited successor w of v, it updates the $rindex$ of v with its smallest visit index ($rindex[v] = min(rindex[v], rindex[w])$) (lines 5, 6), and v is pushed into stack S as v is not the component root ($root = false$) (line 11). (3) If the local root of a node remains unchanged ($root = true$), the component root is found. Then, all members of the SCC are popped out from stack S, and assigned the same component identifier c (lines 7-9). To ensure $rindex[v] \leq rindex[w]$ for next w, which requires that $index < c$ always holds, c and $visitid$ are decreased by 1 and SCC size, respectively (line 10).

Procedure visit2 detects the SCCs involving within E_{s2s}, which is a slight variant of procedure visit1. Given an unvisited head v in E_{s2si} and its $year$ i, it only traverses the unvisited nodes of same-year subgraph using DFS to find the SCCs. (1) It removes the array inGm, and adds the array inGsi that is set to its $year$ i (line 1 of visit1). (2) For each successor w of v, it restricts the year of w to i (line 3 of visit1). Hence, the nodes not in $year$ i and visited by procedure visit1 are both pruned in procedure visit2.

Algorithm staDSCC takes as input the citation graph G, the sets E_{o2n} and E_{s2s} of edges, the max and min year, and outputs the members of each SCC mSCCs, arrays inGm, inGsi and $rindex$. First, inGm, inGsi and $rindex$ are initialized to $false$, 0 and 0 for each node v, respectively (line 1). It then calls procedure visit1 to detect all SCCs reachable from the heads of E_{o2n} for each unvisited head v (lines 2, 3). Besides, for each unvisited head v in E_{s2si}, it calls procedure visit2 to detect all SCCs reachable from the heads of E_{s2si} except those visited by visit1, for each year between min_year and max_year (lines 4-6). Finally, it returns mSCCs, inGm, inGsi and $rindex$, where mSCCs can be assembled by iterating $rindex$ and gathering the nodes by the component identifier (lines 7, 8). Note that all SCCs of citation graph G are detected by only traversing from the edge heads of E_{o2n} and E_{s2s} by Property 1.

We next illustrate staDSCC with an example.

Example 2: We consider the citation graph G with 21 nodes with time attributes ($year$) and 26 edges, shown in Fig. 4. The nodes and edges colored purple and green indicate that they are visited from the edge heads of E_{o2n} and E_{s2s}, respectively. Assume that the nodes are visited in an ascending order, and each node is annotated with its visit index and $rindex$.

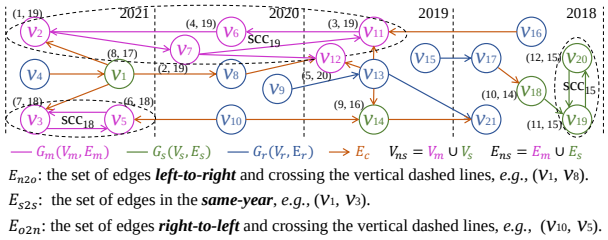


Figure 4. Running example for staDSCC

(1) $E_{o2n} = \{(v_6, v_2), (v_{10}, v_5), (v_{11}, v_6), (v_{16}, v_{11})\}$. It first traverses the head v_2 of edge (v_6, v_2), and the nodes $\{v_2, v_7, v_{11}, v_6, v_{12}\}$ are successively visited by procedure visit1, where SCC_{19} with nodes $\{v_2, v_7, v_{11}, v_6\}$ is detected and assigned with the same identifier 19. SCC_{18} with nodes $\{v_3, v_5\}$ is similarly detected by traversing from v_5. (2) For year 2021, $E_{s2s2021} = \{(v_1, v_2), (v_1, v_3), (v_3, v_5), (v_4, v_1), (v_5, v_3)\}$. Note that, v_2, v_3 and v_5 have been visited from E_{o2n}, and only node v_1 is visited from the head of edge (v_4, v_1). The edge set E_{s2s} of years 2020, 2019 and 2018 is similarly traversed by staDSCC. Finally, SCC_{19}, SCC_{18} and SCC_{15} are the detected SCCs, surrounded by dashed lines. In fact, our staDSCC only visits 12 nodes and traverses 10 edges in the process, instead of the entire graph. □

Correctness. The correctness of algorithm staDSCC can be easily verified from the correctness of Pearce2 [25] and Property 1 of Section III-A.

Time and space complexities. Only the nodes V_{ns} and edges E_{ns} that are reachable from the edge heads of E_{o2n} and E_{s2s} are visited and traversed at most one time. Hence, the time complexity of staDSCC is $\Theta(|V_{ns}| + |E_{ns}|)$, where $|V_{ns}|$ and $|E_{ns}|$ are less than $|V|$ and $|E|$, respectively. staDSCC takes $O(|V| + |E|)$ space, among which the citation graph takes $O(|V| + |E|)$ space, and each of arrays mSCCs, inGm, inGsi and $rindex$ takes $O(|V|)$ space.

Finally, we have proved Theorem 1 by the correctness and complexity analyses of algorithm staDSCC.

Remarks. Although Procedure visit1 of our static algorithm mostly comes from Pearce2 [25], staDSCC is designed based on the properties of citation graphs. It only visits and traverses partial nodes and edges at most once to detect all SCCs in citation graphs, while Pearce2 visits the entire graph in one-pass. Note that, the input E_{o2n}, E_{s2s}, max_year and min_year of staDSCC can be easily obtained when constructing the citation graph G, which does not take extra time.

IV. Analyses of Incremental Detection

Citation graphs are typically large and constantly growing with small changes, *e.g.,* CiteseerX having more than 13 million articles continuously calls a crawling procedure to timely feed new articles into the database with at least 2,000 articles per day [3, 41], and it is impractical to recompute the SCCs from scratch once it gets updated. Further, the incremental SCC detection can be used to support incremental block-wise PageRank computation for article ranking [6] and incremental visual exploration of the citation graph topology [16, 20]. These highlight the need of incremental SCC detection algorithms.

In this section, we analyze key design issues of incremental SCC detection solutions in citation graphs. As scholarly data are rarely updated with deletions [2, 41] and the appends are common for incremental scholarly data analysis tasks [6, 7, 42], we focus on updates in an append manner in this study.

A. Partitioning Citation Graphs

Existing incremental algorithms for SCC detection are for general graphs [28–30, 32], which traverse all the nodes/edges of the entire graph that are reachable from the nodes involved in the updates. However, not all nodes and edges of the citation graphs are needed as shown by Property 1, and Property 2 reveals that traversing the edges in E_{o2n} and E_{s2s} can further improve the efficiency (Section III-A). To do this, we need to partition the citation graph into distinct subgraphs involved with different edge types E_{n2o}, E_{s2s} and E_{o2n}.

Partition. A citation graph $G(V, E)$ is disjointly partitioned into three types of subgraphs $G_m(V_m, E_m)$, $G_s(V_s, E_s)$, $G_r(V_r, E_r)$ and a set E_c of cross edges, where we use the $year$ as the default time granularity, the same setting as existing scholarly databases.

(1) Subgraph $G_m(V_m, E_m)$, where V_m contains the set V_{o2n} of edge heads in E_{o2n} and the set V^+_{o2n} of nodes reachable from V_{o2n}, and $E_m = \{(x, y) | x \in V_m, y \in V_m \ and \ (x, y) \in E\}$. Observe that G_m contains all the SCCs involved with the edges in E_{o2n}.

(2) Subgraph $G_s(V_s, E_s)$ consists of a set of subgraphs (one for each year). $G_s = \{G_{si}(V_{si}, E_{si}) | i \in [min\_year, max\_year]\}$, where min_year and max_year are the minimum and maximum year on citation graphs, respectively. For each $G_{si}(V_{si}, E_{si})$, $V_{si} = V_{s2si} \cup V^+_{s2si} \setminus V_m$, where V_{s2si} is the set of edge heads in E_{s2s} with year i, V^+_{s2si} is the set of nodes with year i reachable from the nodes in V_{s2si}, and $E_{si} = \{(x, y) | x, y \in V_{si} \ and \ (x, y) \in E\}$. Observe that G_s contains all the SCCs involved with the edges in E_{s2s} only.

(3) Subgraph $G_r(V_r, E_r)$, where $V_r = V \setminus V_m \setminus V_s$, and $E_r = \{(x, y) | x, y \in V_r \ and \ (x, y) \in E\}$. Observe that G_r contains all the nodes of G except those in subgraphs G_m and G_s.

(4) Edges $E_c = E \setminus E_m \setminus E_s \setminus E_r$ is a set of cross edges among G_m, G_r, and G_{si} with $i \in [min\_year, max\_year]$. Note that, not all permutations of the three types of subgraphs belong to E_c, as some of them have been defined in G_m, G_s, and G_r, *e.g.,* $x \in G_m$, $y \in G_r$, then edge $(x, y) \in G_m$. Indeed, there are five types of $(x, y) \in E_c$: (a) x in G_r, y in G_m, (b) x in G_{si}, y in G_m, (c) $(x, y) \in E_{n2o}$, x in G_{sj}, y in G_{si} and $i \neq j$, (d) $(x, y) \in E_{n2o}$ *or* E_{s2s}, x in G_r, y in G_{si}, (e) $(x, y) \in E_{n2o}$, x in G_{si}, y in G_r.

Hence, the graph $G(V, E)$ is disjointly partitioned such that $V = V_m \cup V_s \cup V_r$ and $E = E_m \cup E_s \cup E_r \cup E_c$. Note that traversing from the edge heads and tails in the partition can both find all SCCs. However, traversing from the edge heads can reduce the traversal of nodes and edges that do not belong to SCCs, as the edges with 0 in-degree tails are skipped when traversing from the edge heads of E_{o2n} and E_{s2s}.

We next illustrate the partition with an example.

Example 3: Consider Example 2 in Fig. 4 again. After partitioning, G_m, G_s, G_r and E_c are colored purple, green, blue and orange, respectively. Based on the definition of each subgraph, (1) G_m with nodes $\{v_2, v_3, v_5, v_6, v_7, v_{11}, v_{12}\}$ is first determined as they are reachable from the set $V_{o2n} = \{v_2, v_5, v_6, v_{11}\}$ of edge heads in E_{o2n}. (2) G_s consists four subgraphs from year 2018 to 2021: $G_s = \{G_{s2021} \cup G_{s2020} \cup G_{s2019} \cup G_{s2018}\}$, in which G_{s2018} has nodes $\{v_{18}, v_{19}, v_{20}\}$ and edges $\{(v_{18}, v_{19}), (v_{19}, v_{20}), (v_{20}, v_{19})\}$ as they all reachable from the edge heads of same-year edges E_{s2s} with *year* 2018. (3) G_r consists the remaining nodes and their connected edges with nodes $\{v_4, v_8, v_9, v_{10}, v_{13}, v_{15}, v_{16}, v_{17}, v_{21}\}$ and edges $\{(v_9, v_{13}), (v_{13}, v_{21}), (v_{15}, v_{17})\}$. And (4) E_c is a set of cross edges, with edges $\{(v_1, v_2), (v_1, v_3), (v_1, v_8), (v_4, v_1), (v_8, v_{12}), (v_{10}, v_5), (v_{10}, v_{14}), (v_{13}, v_{11}), (v_{13}, v_{12}), (v_{13}, v_{14}), (v_{14}, v_{21}), (v_{16}, v_{11}), (v_{17}, v_{18})\}$. □

We next build connections between the partition and SCCs, which shall be utilized for the design of incremental methods.

Proposition 1: *Non-singleton* SCCs *exist in G_m and G_s only.* □

Proof: From Property 1, all SCCs can be detected by traversing edges in E_{o2n} and E_{s2s} only.

First, for each edge $e \in E_{o2n}$ belonging to an SCC, following the definition of G_m all reachable nodes from the head of e also belong to G_m, and hence the SCC containing e is naturally defined in G_m. Second, for each edge $e \in E_{s2s}$ belonging to an SCC, there are two cases in total. (a) The SCC contains edges of E_{s2s}, E_{o2n} and E_{n2o}. (b) The SCC only contains edges of E_{s2si}. For case (a), the SCC has been defined in G_m. For case (b), following the definition of G_{si}, all reachable nodes from the head of e in the same-year subgraph also belong to G_{si}, and hence the SCC containing e must be defined in G_{si}. To conclude, non-singleton SCCs exist in G_m and G_s only. □

Proposition 1 tells us that only G_m and G_s are needed for detecting all the non-singleton SCCs. Moreover, the edges of all SCCs reachable from the edge heads of E_{o2n} and E_{s2s} belong to E_m and E_s, respectively, and $E_m \cup E_s \cup E_r \cup E_c = E_{n2o} \cup E_{s2s} \cup E_{o2n} = E$.

Proposition 2: *Algorithm* staDSCC *also partitions the citation graph G into G_m, G_s, G_r and E_c.* □

Proof: The partition and algorithm staDSCC are both designed based on the reachability with the same source nodes and pruning conditions. First, staDSCC traverses from all edge heads of E_{o2n} on the entire graph via DFS, hence all nodes of G_m are found by staDSCC. Second, for each year i, it traverses from the edge heads of E_{s2si} on the unvisited nodes of same-year subgraphs via DFS, hence all nodes and edges of G_{si} are also found by staDSCC. Third, G_r and E_c can be easily obtained after G_m and each $G_{si} \in G_s$ are found. □

Proposition 2 tells us the result of staDSCC can be utilized to design incremental algorithms.

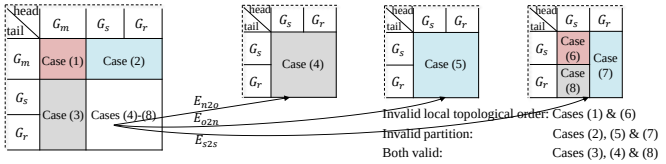


Figure 5. All cases of edge insertions

Partition maintenance. Arrays inGm, inGsi and mSCCs of staDSCC are enough to represent the partition based on the proof of Proposition 2, which shall be updated to maintain the partition for continuous increasing nodes and edges. For the node insertion, it must belong to G_r, and the partition remains unaffected. For the edge insertion (x, y), based on the edge type and the subgraphs to which its tail and head belong, there are the following 8 Cases to maintain the partition, as shown in Fig. 5.

Case (1): $x, y \in G_m$ for any edge type;
Case (2): $x \in G_m$ and $y \in G_s \cup G_r$ for any edge type;
Case (3): $x \in G_s \cup G_r$ and $y \in G_m$ for any edge type;
Case (4): $x, y \in G_s \cup G_r$ with $(x, y) \in E_{n2o}$;
Case (5): $x, y \in G_s \cup G_r$ with $(x, y) \in E_{o2n}$;
Case (6): $x, y \in G_s$ with $(x, y) \in E_{s2s}$;
Case (7): $x \in G_s \cup G_r$ and $y \in G_r$ with $(x, y) \in E_{s2s}$;
Case (8): $x \in G_r$ and $y \in G_s$ with $(x, y) \in E_{s2s}$.

We further conduct the Propositions for maintaining the partition for these Cases.

Proposition 3: *The citation graph partition only needs to be adjusted when inserting edge $(x, y) \in$ Cases (2), (5) & (7).* □

Proof: Proposition 3 can be proved by the definition of the citation graph partition where Cases (1), (3), (4), (6) & (8) of edge insertions naturally belong to G_m, G_s, G_r and E_c and do not need to be adjusted. For Case (2), based on the reachability of the nodes in G_m, the node y should be adjusted such that y belongs to G_m. For Cases (5) & (7), the partition should also be adjusted by the definition of subgraphs G_m, G_s, respectively. □

This tells us that the citation graph partition remains unaffected for the other five Cases of edge insertions: Cases (1), (3), (4), (6) & (8), and the new SCCs by merging different subgraphs must not be introduced by these edge insertions.

Proposition 4: *When inserting edge $(x, y) \in$ Case (7), the partition is maintained by only marking y in G_s.* □

Proof: We show this from the definition of citation graph partition. First, G_m remain unchanged as the inserted edge $(x, y) \in E_{s2s}$. Second, based on the definition of G_{si}, the node y and its reachable nodes of the same-year subgraph should be marked in G_{si} to maintain the partition. However, it is easy to prove by contradiction that all edges of G_r are in E_{n2o}. That is there are no edges of E_{s2s} belonging to G_r that need to be traversed from y. And hence, only y is marked in G_s to maintain the partition. □

Proposition 4 is used to maintain the partition.

Remarks. Note that different from the partition for general graphs [43], our partition is designed based on the properties of citation graphs.

B. Maintaining SCCs

In addition to the partition maintenance for citation graphs, efficient maintenance of SCCs is also needed for incremental algorithms, which involves two main operations. First, SCCs may merge with each other to form bigger SCCs during the update process. Second, it needs to efficiently determine whether two nodes are in the same SCC. We next depict how to efficiently maintain SCCs.

The data structure disjoint set [23] is utilized to represent the nodes of SCCs. Specifically, it maintains a *dummy node* for each SCC and the *dummy node* represents the members of the SCC, and we use dx to represent the dummy node of the SCC to which node x belongs. Two operations can be implemented by functions union and find of disjoint sets, respectively. (1) union(dx, dy): given two different *dummy nodes* dx and dy, it merges the two sets (*i.e.,* SCCs) containing nodes x and y, respectively, and makes dx as the dummy node of the new set (merged SCC). (2) find(x): given a node x, it returns its dummy node dx. Both functions take $O(1)$ amortized time [23]. Once SCCs are represented by dummy nodes, we naturally have a DAG representation of the citation graph.

For incremental algorithms, SCCs are essentially detected on DAGs, which are maintained during the process. Note that after static algorithm staDSCC detects the SCCs in the original citation graph, we have generated the DAG, represented by the disjoint set, *i.e.,* dSet. And dSet is initialized by the output $rindex$ of staDSCC. In the following, the node with a prefix d stands for the dummy nodes of its represented SCC, by using find function of the disjoint set dSet.

C. Local Topological Order

We finally introduce local topological orders to reduce graph traversal costs. As explained before, incremental detection of SCCs is performed on the DAG representations of citation graphs. Indeed, there is a close connection between the topological order and the DAG representation, as shown below.

Proposition 5: *A directed graph is a* DAG *if and only if it has a topological order [23].* □

Proposition 5 reveals that no extra SCCs are introduced when inserting edge (x, y) has a valid topological order (*i.e.,* $ord(x) \prec ord(y)$), as the updated citation graph remains a DAG. That is, only the edges with invalid topological orders need to be handled.

Further, by Proposition 1, we only need to deal with the topological orders of the DAG representations of subgraphs G_m and each G_{si}, instead of the entire DAG, which are referred to as *local topological orders*.

We next give an analysis of local topological orders to facilitate the design of incremental algorithms.

Proposition 6: *Static algorithm* staDSCC *produces a local topological order of the citation graph G.* □

Proof: We show this by combining the properties of Pearce2. As illustrated, we have generated the DAG representation of the graph after SCCs are detected. First, the topological order of the DAG is also a byproduct of Pearce2. Specifically, it assigns $rindex$ a non-increasing order for each node following the backtracking order of DFS and the nodes in the same SCC share the same order (lines 7-10 of procedure visit1 of staDSCC). That is the $rindex$ of each node can also be utilized to represent the topological order of the DAG representation of the graph. Second, all nodes and edges of G_m and G_{si} are visited by algorithm staDSCC, which is designed on the basis of Pearce2 without changing its logic. Thus, staDSCC produces the topological orders of the DAG representations of G_m and each G_{si}. That is, staDSCC produces the local topological order of G. □

Proposition 6 reveals that staDSCC produces a local topological order ($rindex$), and can be utilized to design incremental algorithms.

Proposition 7: *Only when inserting edge (x, y) into a maintained partition with $(x, y) \in$ Cases (1) & (6) that violates the local topological order, the local topological order needs to be adjusted.* □

Proof: Proposition 7 can be easily obtained by the definition of the local topological order as the citation graph partition is already maintained. □

This tells us that the local topological order of G remains valid and unaffected for Cases (3), (4) & (8).

Local topological order maintenance. The local topological order is simply maintained for the dummy nodes only, *i.e.,* the DAG representations of subgraphs G_m and each $G_{si} \in G_s$. Indeed, all members of an SCC represented by a dummy node have the same order. Similar to [28, 30, 31, 33, 34, 44], we use a set of ordered lists oLists to flexibly and efficiently reorder the local topological order, which is initialized by the output $rindex$ of staDSCC. There are in total $(max\_year - min\_year + 2)$ ordered lists, where each ordered list maps a dummy node to a unique *order* to represent the topological order of its DAG. Besides, the number of orders in oLists always equals the number of dummy nodes in G_m and G_s, and for each newly created order, the old order is first deleted using the delete function of ordered lists.

V. Dealing with Single Updates

In this section, we present our incremental method to deal with single updates, as most existing incremental methods on general graphs are for single updates [28–30, 35, 37], and existing batch methods [37], [29] are based on single updates [35], [29], respectively. Based on the previous analysis, our single update method detects the SCCs by discovering the nodes bounded by the minimum cover of affected node pairs and affected edges on G_m and G_s to both maintain the partition and local topological order.

The main result is stated below.

Theorem 2: *Given the original partition and local topological order of citation graph $G(V, E)$ and single node/edge insertions, there exists a bounded incremental algorithm that detects the* SCCs *and maintains the partition and local topological order of the updated G in $O(\|\mathsf{AFF}\| log \|\mathsf{AFF}\| + |\mathsf{AFFE_m}| + |\mathsf{AFFE_s}|)$ time, where (1)* AFF *is bounded,* i.e., $\|\mathsf{AFF}\| \leq 2\|K_{min}\|$ *such that K_{min} is the minimum cover of affected node pairs, and (2)* $\mathsf{AFFE_m}$ *and* $\mathsf{AFFE_s}$ *are the affected edges on G_m and on G_s, respectively.* □

A. Overview of Single Updates

We first discuss the types of single updates that incremental methods need to handle based on the analyses of incremental detection. Note that we handle insertions only in this study, as scholarly data are rarely updated with deletions [2, 41].

(1) Single node insertions. When inserting a node x, it must belong to G_r, and the partition and local topological order remain valid by Propositions 1, 3 and 7. Hence, x is simply marked in G_r.

(2) Single edge insertions. When inserting an edge (x, y), the Cases (1)–(8) analyzed in Section IV are grouped into 4 cases by considering whether x and y in G_m, G_s, G_r or not, also illustrated in Fig. 5.

(a) From G_m to G_m. (*i.e.,* Case (1)). The partition is always valid, and the local topological order is invalid if $ord(x) \succ ord(y)$ by Propositions 3 and 7, which needs to be maintained based on [23].

(b) From G_m to $G_s \cup G_r$. (*i.e.,* Case (2)). The partition is invalid by Proposition 3, and the partition and local topological order need to be maintained.

(c) From $G_s \cup G_r$ to G_m. (*i.e.,* Case (3)). The partition and local topological order are both valid, and no extra SCCs are introduced by Propositions 1, 3 and 7. Indeed, there is nothing to do for this case.

(d) From $G_s \cup G_r$ to $G_s \cup G_r$. (*i.e.,* Cases (4)–(8)). For Cases (4) & (8), the partition and local topological order are both valid, which is the same as case (c). For Cases (5) & (7), the partition is invalid, which is the same as case (b). For Case (6), the partition is always valid, and the local topological order is invalid if $ord(x) \succ ord(y)$, which is the same as case (a).

After inserting edge (x, y), we say the edge is valid (resp. invalid) *w.r.t.* the partition if the partition is valid (resp. invalid), and similarly, we say the edge is valid (resp. invalid) *w.r.t.* the local topological order if the local topological order is valid (resp. invalid). The SCCs are affected and need to be detected when the partition or local topological order is invalid in the above cases, where the partition is firstly maintained, followed by the local topological order. Observe also that the nodes and edges of G_m increase only, while the nodes and edges of G_{si}, G_r and E_c may increase or decrease.

Data structures for single updates. Shared (used for all cases) and private (used for one case alone) data structures are included for single updates. Five shared data structures are utilized based on the analyses of Section IV: arrays mSCCs, inGm and inGsi to represent the partition, disjoint set dSet to represent the partition of the nodes defined by SCCs, and oLists to represent the local topological order. These five data structures are either the output of the static algorithm staDSCC or are initialized by it. The private data structures will be introduced when handling each case.

Input: Edge (x, y) with $x, y \in G_m$, citation graph G, arrays mSCCs, inGm, inGsi, ordered lists oLists, disjoint set dSet.
Output: Updated G, inGm, inGsi, mSCCs, oLists, dSet.

1. Update G by inserting (x, y);
2. **if** $dx = dy$ **or** oLists$[G_m]$.order(dx, dy) **then return**
3. AFF $= \emptyset$; $isExist = false$;
4. $inF[dv] = inB[dv] = false$ for all $v \in G_m$;
5. discover(dx, dy); maintain$(dx, dy, isExist,$ AFF$)$;
6. **return** updated G, mSCCs, inGm, inGsi, oLists, dSet.

Procedure discover(dx, dy)

1. $forwPQ = \{dy\}$; $backPQ = \{dx\}$;
2. $f = forwPQ.top()$; $b = backPQ.top()$;
3. $numFE = f$.outDegree; $numBE = b$.inDegree;
4. **while** oLists$[G_m]$.order(f, b) **or** $f = b$ **do**
5. $numFE$ and $numBE$ are decreased by their smaller ones;
6. **if** $numFE = 0$ **then**
7. AFF $\cup = \{f\}$; $forwPQ.pop()$;
8. **for each** out edge (f, z) **do**
9. **if** $inB[dz]$ **then** /* SCC is detected */
10. $isExist = true$; $forwPQ.push(dz)$; $inF[dz] = true$;
11. **if** $!inF[dz]$ **then**
12. $forwPQ.push(dz)$; $inF[dz] = true$;
13. **if** $forwPQ = \emptyset$ **then** $f = dx$; **else** $f = forwPQ.top()$;
14. $numFE = f$.outDegree;
15. **if** $numBE = 0$ **then**
16. Do backward search similar to forward search (lines 7-14).

Procedure maintain$(dx, dy, isExist,$ AFF$)$

1. Compute rTO and $ceiling[dv]$ for all nodes in AFF;
2. **if** $isExist$ **then**
3. Maintain mSCCs and dSet for SCCs;
4. **for each** dv **in** rTO **do**
5. oLists$[G_m]$.insertBefore$(ceiling[dv], dv)$.

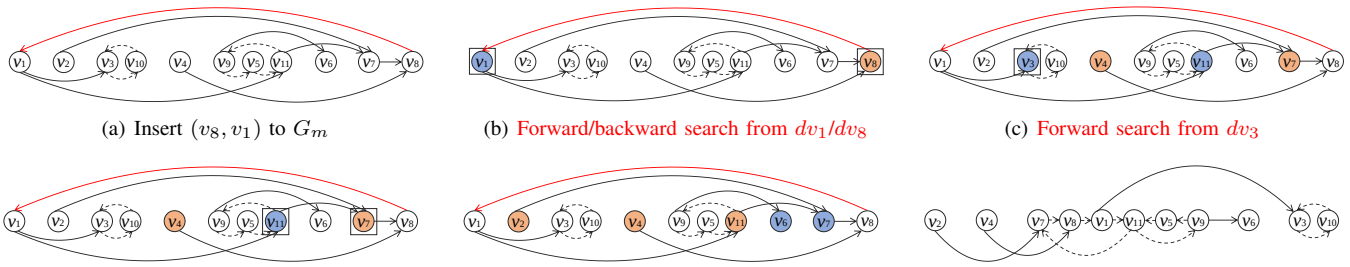
Figure 6. Algorithm incGm2Gm

B. Case G_m to G_m

We then present incremental algorithm incGm2Gm to handle case G_m to G_m with invalid local topological orders (*i.e.,* Case (1)) based on the analysis in Section IV.

Algorithm incGm2Gm essentially performs on the DAG representation of G_m, shown in Fig. 6. It detects the SCC and maintains the local topological order to traverse the least amount of nodes, *i.e.,* the visited nodes are bounded by the minimum cover of affected node pairs of edge insertion. It operates in two procedures discover and maintain, where discover checks whether new SCCs exist and finds the dummy nodes with invalid local topological orders, and maintain detects new SCC if exist and maintains the local topological orders for the nodes found by discover. The details of incGm2Gm is shown below.

Private data structures in incGm2Gm**.** In addition to five shared data structures, the following private data structures are needed for incGm2Gm. (1) Array AFF stores the dummy nodes having invalid local topological orders, (2) arrays inF and inB indicate whether a dummy node is visited by forward or backward search, respectively, for all dummy nodes in G_m.



(a) Insert (v_8, v_1) to G_m (b) Forward/backward search from dv_1/dv_8 (c) Forward search from dv_3

(d) Forward/backward search from dv_{11}/dv_7 (e) Procedure discover finished (f) Procedure maintain finished

Figure 7. Running example for incGm2Gm (All nodes and edges belong to G_m)

Procedure discover checks whether new SCCs exist and finds the dummy nodes with invalid local topological orders AFF by iteratively accessing the successors of the inserted edge head and predecessors of the inserted edge tail. Given input dummy nodes dx and dy, it updates AFF, inF, inB, $isExist$ (indicating whether a new SCC is introduced or not).

(1) First, min priority queue $forwPQ$ and max priority queue $backPQ$ are created, which are initialized to contain dy and dx, and store the forward search dummy nodes (*i.e.,* the successors of dy) and the backward search dummy nodes (*i.e.,* the predecessors of dx) (line 1). Variables f and b are the top nodes of $forwPQ$ and $backPQ$, respectively (line 2), and $numFE$ and $numBE$ are the out-degree of f and the in-degree of b, respectively (line 3). (2) Then it recursively identifies the set AFF of nodes with invalid local topological orders from those nodes reachable to dx or from dy (lines 4-16). $numFE$ and $numBE$ are decreased by their smaller ones, which determines to perform forward or backward search (line 5). (3) If the current $numFE$ is equal to or smaller than $numBE$, forward search is performed (lines 7-14). (a) The forward search node f violates the local topological order, which is added into AFF, and removed from $forwPQ$ (line 7). (b) The out-neighbors z of f are then processed and dz is the dummy node to which SCC it belongs (lines 8-13). If dz is already visited by backward search, $isExist$ is set to $true$ as there exists an SCC, dz is pushed into $forwPQ$, and $inF[dz]$ is set to $true$ (lines 9, 10). If dz is not visited by forward search, dz is simply pushed into $forwPQ$, and $inF[dz]$ is set to $true$ (lines 11, 12). (c) If $forwPQ$ is empty, f is set to dx, which terminates the while loop, and f is also set to the top of $forwPQ$, otherwise (line 13). (d) $numFE$ is updated to the out-degree of the current f (line 14). (4) The backward search is performed along the same lines as the forward search if the current $numBE$ is equal to or smaller than $numFE$ (lines 15, 16).

Procedure maintain detects new SCCs if exist, and maintains local topological orders for the nodes in AFF found by discover, using the forward and backward DFS on the dummy nodes of AFF. Given input dummy nodes dx, dy, $isExist$ and AFF, it outputs the updated mSCCs, dSet and oLists$[G_m]$ with valid topological orders.

(1) It first computes rTO and the $ceiling$ of each dummy nodes of AFF using DFS based on [23], where rTO is the reverse local topological orders of the dummy nodes in the connected subgraph induced by AFF, and $ceiling[dv]$ is the dummy node not in AFF with the lowest order when performing DFS on the dummy nodes of AFF, where backtracing is triggered once it meets nodes not in AFF (line 1). (2) If a new SCC is introduced, mSCCs and dSet are maintained using DFS as well (lines 2, 3). The disjoint set dSet is updated by its operation union (dx, dy), and mSCCs is updated by merging the SCCs connected by dx and dy into a new SCC. Note that inserting edge (x, y) may lead to the merging of more than two SCCs. (3) For each dummy node dv in rTO, its local topological order is maintained by the insertBefore function of ordered lists (lines 4, 5). In fact, only the $ceiling$ of dv in AFF reachable from dy need to be computed, as the $ceiling[dv]$ for all nodes dv in AFF reachable to dx are $ceiling[dy]$. Besides, the rTO of nodes dv for each node in AFF reachable from dy is obtained when computing $ceiling[dv]$ using DFS, and the rTO for the nodes dv in AFF reachable to dx is computed using backward search.

Algorithm incGm2Gm takes as input an edge (x, y), citation graph G, arrays mSCCs, inGm, inGsi, ordered lists oLists, disjoint set dSet, and returns the updated G, mSCCs, inGm, inGsi, oLists, dSet. (1) It first updates G by inserting (x, y), and finds the dummy nodes dx and dy (line 1). (2) If $dx = dy$, nodes x and y are in the same SCC. If oLists$[G_m]$.order(dx, dy) holds, the local topological order remains valid, which means G_m remains a DAG by Proposition 5. Hence, nothing needs to be done in these cases (line 2). (3) If (dx, dy) violates the local topological order of G_m, it initializes AFF to empty, $isExist$ to false, inF and inB to $false$ for all dummy nodes of G_m, and calls procedures discover and maintain to detect the SCC (lines 3-5). (4) Finally, it returns the updated G, inGm, inGsi, mSCCs, oLists and dSet (line 6).

We illustrate incGm2Gm with an example below.

Example 4: Consider inserting edge (v_8, v_1) into G_m with 11 nodes and 13 edges, shown in Fig. 7. Algorithm incGm2Gm is actually performed on the dummy nodes of G_m, where the inner edges of SCCs are indicated by dashed lines. The blue nodes stand for the dummy nodes visited by forward search, and are in $forwPQ$. The orange nodes stand for the dummy nodes visited by backward search, and are in $backPQ$. The forward or backward search is performed from the dummy nodes surrounded by a box, oLists of G_m saves the topological orders of its dummy nodes, *i.e.,* from left to right and in an increasing order, dSet maintains the representative dummy nodes of each SCC, *e.g.,* dv_1, dv_3, mSCCs stores the members of each SCC, *i.e.,* $\{\{v_3, v_{10}\}, \{v_5, v_9, v_{11}\}\}$, and inGm and

inGsi remain unchanged.

As edge (v_8, v_1) violates the local topological order, incGm2Gm goes to line 3, and calls procedures discover and maintain, shown in Fig. 7(a). Procedure discover iteratively accesses the successors of dv_1 and predecessors of dv_8 to update $isExist$ and AFF. Specifically, (1) dv_1 and dv_8 are pushed into $forwPQ$ and $backPQ$, and variables f and b are set to dv_1 and dv_8, respectively. It performs both froward and backward search from dv_1 and dv_8, respectively, as $ord(dv_1) \prec ord(dv_8)$ and $numFE = numBE = 0$, shown in Fig. 7(b). (2) The visited nodes dv_1 and dv_8 are removed from $forwPQ$ and $backPQ$ and both pushed into AFF, respectively. Beside, out-neighbors $\{dv_3, dv_{11}\}$ of dv_1 and in-neighbors $\{dv_7, dv_4\}$ of dv_8 are pushed into $forwPQ$ and $backPQ$, respectively. Variables f and b are updated to dv_3 and dv_7, respectively, and it only performs froward search as $numFE = 0$, $numBE = 2$, shown in Fig. 7(c). (3) Similar to (2), dv_3, dv_{11} and dv_7 are then processed and pushed into AFF, shown in Fig. 7(d). (4) After procedure discover finished, it finds $\mathsf{AFF} = \{dv_1, dv_8, dv_3, dv_{11}, dv_7\}$ and $isExist = true$, shown in Fig. 7(e). Procedure maintain detects the new SCC with dummy nodes $\{dv_1, dv_8, dv_{11}, dv_7\}$ from AFF, maintains the local topological order and updates oLists, mSCCs, shown in Fig. 7(f). □

The correctness of incGm2Gm is assured as follows.

Theorem 3: *Algorithm* incGm2Gm *correctly detects the* SCCs *and maintains the partition and local topological order for case G_m to G_m. Further,* AFF *is bounded,* i.e., $\|\mathsf{AFF}\| \leq 2\|K_{min}\|$, *where K_{min} is the minimum cover of affected node pairs.* □

Proof: It is trivial for (x, y) in the same SCC or with a valid local topological order. We consider (x, y) violates the local topological order, and show this from (1) AFF found by algorithm incGm2Gm is a cover of affected node pairs, (2) $\|\mathsf{AFF}\| \leq 2\|K_{min}\|$, where K_{min} is the minimum cover of affected node pairs, (3) algorithm incGm2Gm correctly detects the SCCs if exists, and (4) it correctly maintains the partition and local topological order. Note that, we only consider the DAG representation of G_m, and dx, dy are the dummy nodes of x, y, respectively.

(1) We first show AFF is a cover of affected node pairs. Let $V_f = \{s | dy \rightsquigarrow s \wedge (ord(s) \prec ord(dx) \vee s = dx)\}$, and $V_b = \{s | s \rightsquigarrow dx \wedge (ord(s) \succ ord(dy) \vee s = dy)\}$, where $dy \rightsquigarrow s$ and $s \rightsquigarrow dx$ denote there exist paths from dy to s and from s to dy on the dummy nodes of G_m.

It only needs to show for all paths $s \rightsquigarrow t$ with $s \in V_b, t \in V_f$, and $s \notin \mathsf{AFF} \wedge t \notin \mathsf{AFF}$, then $ord(s) \prec ord(t)$ holds, as the other paths $s \rightsquigarrow t$ on the dummy nodes of G_m are not affected by (x, y).

We divide AFF into AFF_f and AFF_b, where $\mathsf{AFF}_f = \{s \in \mathsf{AFF} | dy \rightsquigarrow s\}$ and $\mathsf{AFF}_b = \{s \in \mathsf{AFF} | s \rightsquigarrow dx\}$. That is to show for $\forall s \in V_b \setminus \mathsf{AFF}_b$ and $\forall t \in V_f \setminus \mathsf{AFF}_f$, then $ord(s) \prec ord(t)$ holds.

Let s_h be the node in $V_b \setminus \mathsf{AFF}_b$ with the highest order, and t_l be the node in $V_f \setminus \mathsf{AFF}_f$ with the lowest order. According to the termination condition of procedure discover (line 4), $ord(t_l) \succ ord(s_h)$ holds. Hence, for $\forall s \in V_b \setminus \mathsf{AFF}_b$ and $\forall t \in V_f \setminus \mathsf{AFF}_f$, $ord(s) \prec ord(s_h) \vee s = s_h$ and $ord(t_l) \prec ord(t) \vee t_l = t$ always holds, which means $ord(s) \prec ord(t)$.

(2) We then show $\|\mathsf{AFF}\| \leq 2\|K_{min}\|$, where K_{min} is the minimum cover of affected node pairs. AFF is also divided into AFF_f and AFF_b. Actually, either AFF_f or AFF_b is the minimum cover of affected node pairs, as for $\forall t \in \mathsf{AFF}_f$, $\forall s \in \mathsf{AFF}_b$, pairs (s, t) are the affected node pairs. Note that, incGm2Gm balances the out-degrees of forward search (*i.e.,* AFF_f) and in-degrees of backward search (*i.e.,* AFF_b) in procedure discover (line 5).

There are three cases in total. (a) If the last search is forward before procedure discover terminates, then there exist some predecessors of nodes in backward are not searched, which means $\|\mathsf{AFF}_b\| < \|\mathsf{AFF}_f\|$. We assume the predecessors of node s in backward are not searched, *i.e.,* $s \notin \mathsf{AFF}_b$. Then for $\forall t \in \mathsf{AFF}_f$, $ord(s) \succ ord(t) \vee s = t$ holds for all paths $s \rightsquigarrow t$, *i.e.,* pairs (s, t) are the affected node pairs. However, $s \notin \mathsf{AFF}_b \wedge t \notin \mathsf{AFF}_b$ means AFF_b is not a cover, *i.e.,* AFF_f is the minimum cover. Hence, $\|\mathsf{AFF}\| = \|\mathsf{AFF}_f\| + \|\mathsf{AFF}_b\| < 2\|\mathsf{AFF}_f\| = 2\|K_{min}\|$. (b) If the last search is backward before procedure discover terminates, we have $\|\mathsf{AFF}_f\| < \|\mathsf{AFF}_b\|$, and AFF_b is the minimum cover, similar to (a). Hence, $\|\mathsf{AFF}\| = \|\mathsf{AFF}_f\| + \|\mathsf{AFF}_b\| < 2\|\mathsf{AFF}_b\| = 2\|K_{min}\|$. (c) If the last search is forward and backward before procedure discover terminates, then $\|\mathsf{AFF}_f\| = \|\mathsf{AFF}_b\|$, *i.e.,* AFF_f or AFF_b is the minimum cover. Hence, $\|\mathsf{AFF}\| = \|\mathsf{AFF}_f\| + \|\mathsf{AFF}_b\| = 2\|\mathsf{AFF}_b\| = 2\|\mathsf{AFF}_f\| = 2\|K_{min}\|$. Thus, $\|\mathsf{AFF}\| \leq 2\|K_{min}\|$ always holds.

(3) We next show incGm2Gm correctly detects the SCCs if exists. Assume there exists a new SCC after inserting edge (x, y). Based on the definition of the cover of affected node pairs, for each member s of the new SCC, (s, s) is an affected node pair, hence, s must belong to the cover. That means all members of the new SCC belong to AFF. It is easy to perform DFS on the nodes of AFF to detect all members of the SCC if (x, y) introduces a new SCC.

(4) We finally show incGm2Gm correctly maintains the partition and local topological order. (a) The partition remains valid, as x and y both belong to G_m by Proposition 3. (b) The topological order of each G_{si} remains valid. (c) We next consider the topological order of G_m. Actually, it remains to only maintain the order for each node of AFF, as the other nodes of G_m are already with valid topological orders based on the definition of the cover of affected node pairs. Following the topological orders of nodes in AFF, we can easily create an order for each node lower than its *ceiling* by the insertBefore function of the ordered list as shown in procedure maintain. Hence, algorithm incGm2Gm correctly maintains the topological order of G_m.

Putting these together, we have proved Theorem 3. □

Time complexity. For single edge insertions, algorithm incGm2Gm takes $O(\|\mathsf{AFF}\| log \|\mathsf{AFF}\|)$ time, where AFF is a cover of the affected node pairs of edge insertions. Note that,

Input: Edge (x, y) with $x \in G_m$ and $y \in G_{si} \cup G_r$, citation graph G, arrays mSCCs, inGm, inGsi, ordered lists oLists, disjoint set dSet.
Output: Updated G, inGm, inGsi, mSCCs, oLists, dSet.

1. Update G by inserting (x, y);
2. $v_{lo} = +\infty$; $Rdy = \emptyset$; $isVisit[dv] = false$ for $\forall v \in G_s \cup G_r$;
3. scanGsGr1(dy, v_{lo});
4. Update inGm and create orders for all nodes in Rdy;
5. **if** oLists$[G_m]$.order(dx, v_{lo}) **then return** ;
6. incGm2Gm$(dx, dy, G,$ inGm, inGsi, mSCCs, oLists, dSet);
7. **return** updated G, mSCCs, inGm, inGsi, oLists, dSet.

Procedure scanGsGr1(dy, v_{lo})

1. $isVisit[dy] = true$;
2. **for each** out edge (dy, z) **do**
3. **if** $dz \in G_m$ **and** oLists$[G_m]$.order(dz, v_{lo}) **then** $v_{lo} = dz$;
4. **if** $!isVisit[dz]$ **and** $dz \in G_s \cup G_r$ **then** scanGsGr1(dz, v_{lo});
5. $Rdy.push(dy)$;

Figure 8. Algorithm incGm2Gsr

algorithm incGm2Gm is bounded by the minimum cover of affected node pairs K_{min} as $\|\mathsf{AFF}\| \leq 2\|K_{min}\|$, which takes the least amount of work when the out or in-neighbors of the cover are required to be traversed.

The time cost arises from procedures discover and maintain. (1) In procedure discover, at most $\|\mathsf{AFF}\|$ dummy nodes are inserted and removed from the min or max priority queue. Each node in $\|\mathsf{AFF}\|$ is visited at most once by the operation of the disjoint set, which takes $O(\|\mathsf{AFF}\|)$ time. Hence, procedure discover takes $O(\|\mathsf{AFF}\|\ log\|\mathsf{AFF}\|)$ time in total, where the log factor arises from the use of the priority queue. (2) In procedure maintain, it takes $O(\|\mathsf{AFF}_f\|)$ time to compute rTO and $ceiling[dv]$ for each node dv reachable from dy in AFF (*i.e.,* AFF_f) using DFS, $O(\|\mathsf{AFF}_b\|)$ time to compute rTO of the nodes reachable to dx in AFF (*i.e.,* AFF_b) using backward DFS, and $O(|\mathsf{AFF}|)$ time to maintain mSCCs and dSet, as each operation on disjoint sets takes $O(1)$ time. Besides, at most $|\mathsf{AFF}|$ orders are created to maintain the local topological order, which takes $O(|\mathsf{AFF}|)$ time.

C. Case G_m to $G_s \cup G_r$

We next present incremental algorithm incGm2Gsr to handle case G_m to $G_s \cup G_r$ with the invalid partition (*i.e.,* Case (2)) based on the analysis in Section IV.

Algorithm incGm2Gsr essentially performs on the DAG representation of G, shown in Fig. 8. It first maintains the partition using procedure scanGsGr1 to scan on the subgraph $G_s \cup G_r$, and then detects the SCCs and maintains the local topological with the help of algorithm incGm2Gm. The details of incGm2Gsr is shown below.

Private data structures & variables in incGm2Gsr**.** (1) Array Rdy stores the dummy nodes of $G_s \cup G_r$ reachable from dy with reverse local topological orders, (2) array $isVisit$ indicates whether a dummy node of $G_s \cup G_r$ is visited by procedure scanGsGr1 or not, and (3) variable v_{lo} stores the dummy node in G_m with the lowest order when performing DFS on $G_s \cup G_r$, where backtracing is triggered once the nodes in G_m are met.

Procedure scanGsGr1 finds all reachable dummy nodes from dy on $G_s \cup G_r$. Given input variables dy, v_{lo}, it updates v_{lo}, Rdy, and $isVisit$.

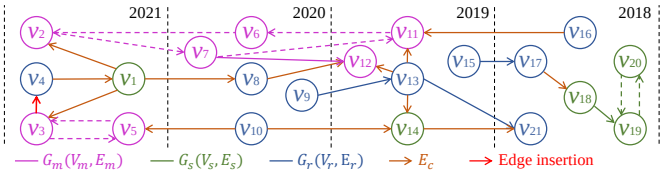


Figure 9. Running example for incGm2Gsr

(1) First, $isVisit[dy]$ is set to $true$ (line 1). (2) The out-neighbors z of dy are then processed recursively, and dz is the dummy node to which SCC node z belongs (lines 2-4). If $dz \in G_m$ and oLists$[G_m]$. order(dz, v_{lo}) holds, v_{lo} is updated to dz with a lower topological order, and the search in G_m is not needed (line 3). If dz is not visited, and $dz \in G_s \cup G_r$, it recursively calls scanGsGr1 with dz and v_{lo} (line 4). (3) After all out-neighbors of dy are visited, dy is pushed into Rdy such that it stores the reverse local topological orders of the reachable dummy nodes of $G_s \cup G_r$ (line 5).

Algorithm incGm2Gsr takes the same input and output as algorithm incGm2Gm, except that $x \in G_m$ and $y \in G_{si} \cup G_r$.

(1) It first updates G by inserting (x, y), finds dummy nodes dx, dy, and initializes v_{lo} to $+\infty$, Rdy to empty, $isVisit$ to $false$ for all dummy nodes of $G_s \cup G_r$ (lines 1-2). (2) It then calls procedure scanGsGr1 with dy and v_{lo} and updates v_{lo}, Rdy and $isVisit$ (line 3). (3) For each dummy node dv in Rdy, it updates $inG_m[v] = true$ for each v with dummy node dv to maintain the partition. Besides, it creates a decreasing order using insertBefore that is lower than v_{lo} for each dv in Rdy (line 4). That is, the topological order of G_m without edge (dx, dy) is maintained [23]. (4) If oLists$[G_m]$.order(dx, v_{lo}) holds, then oLists$[G_m]$.order(dx, dy) holds as the topological order of each dummy node dv in Rdy is lower than v_{lo} and higher than dx by the insertBefore function. That is, the local topological order of G_m remains valid, and G_m remains a DAG by Proposition 5. Hence, nothing needs to be done in this case (line 5). (5) If (dx, dy) violates the local topological order of G_m, it calls incGm2Gm with dx, dy, G, mSCCs, inGm, inGsi, oLists, dSet to detect the SCCs and maintain the local topological order (line 6). (6) Finally, it returns the updated G, inGm, inGsi, mSCCs, oLists, and dSet (line 7).

We next illustrate algorithm incGm2Gsr with an example, shown in Fig. 9.

Example 5: Consider inserting edge (v_3, v_4) with $v_3 \in G_m$, $v_4 \in G_r$ into G along the same setting as Example 2, where the edges of SCCs are drawn with dashed lines. Besides, the topological orders of dummy nodes in G_m have $ord(dv_3) \prec ord(dv_2) \prec ord(dv_{12})$ by the results of staDSCC. Procedure scanGsGr1 is first performed from dv_4, Rdy is $\{dv_8, dv_1, dv_4\}$, and v_{lo} is dv_3. Then $\{v_8, v_1, v_4\}$ are marked in G_m and a decreasing order lower than dv_3 is created for Rdy such that $ord(dv_4) \prec ord(dv_1) \prec ord(dv_8) \prec ord(dv_3) \prec ord(dv_2) \prec ord(dv_{12})$. It calls incGm2Gm to detect the new SCC with $\{dv_3, dv_4, dv_1\}$, and maintain the topological order of G_m such that $ord(dv_3) \prec ord(dv_8) \prec ord(dv_2) \prec ord(dv_{12})$. □

The correctness of incGm2Gsr is assured as follows.

Theorem 4: *Algorithm* incGm2Gsr *correctly detects the* SCCs *and maintains the partition and local topological order for case G_m to $G_s \cup G_r$. Further,* AFF *is bounded,* i.e., $\|\mathsf{AFF}\| \leq 2\|K_{min}\|$, *where K_{min} is the minimum cover of affected node pairs.* □

Proof: We prove the correctness of algorithm incGm2Gsr from it correctly (1) maintains the partition, (2) maintains the local topological order, and (3) detects the SCC if exists. Assume inserting (x, y) with $x \in G_m$, $y \in G_s \cup G_r$, and V_{dm}, V_{dsi}, V_{dr} are the sets of dummy nodes of V_m, V_{si} and V_r, respectively.

(1) Algorithm incGm2Gsr first finds all reachable nodes from the dummy node dy of y on the DAG representation of G, and updates inGm on the original graph. Hence, all reachable nodes from y on G belong to G_m, which maintains the partition of G_m. Besides, the partition of G_{si} and G_r remains valid based on Proposition 3. That is the partition is correctly maintained.

(2) There are three types of edges violating the topological orders on the DAG representation of G_m, *i.e.,* $E_1 = \{(s,t)|s,t \in V_{dsi} \cup V_{dr}\}$, $E_2 = \{(s,t)|s \in V_{dsi} \cup V_{dr}, t \in V_{dm}\}$, $E_3 = \{(dx, dy)\}$. For each $(s,t) \in E_1$, $ord(s) \prec ord(t)$ is easily maintained, as s and t are found by scanGsGr1 (DFS based algorithm) which also produces their topological orders [23]. For each node $s \in V_{dsi} \cup V_{dr}$, $t \in V_{dm}$, v_{lo} is the node with the lowest order of V_{dm}. It creates an order for each node s lower than v_{lo}, hence $ord(s) \prec ord(t)$ holds for each $(s,t) \in E_2$. For (dx, dy) of E_3, $ord(dx) \prec ord(dy)$ is guaranteed by algorithm incGm2Gm.

(3) The SCC is correctly detected by algorithm incGm2Gm if exists.

Putting these together, we have proved Theorem 4. □

Time complexity. Algorithm incGm2Gsr takes $O(\|\mathsf{AFF}\| log\|\mathsf{AFF}\| + |\mathsf{AFFE_m}|)$ time, where AFF is a cover of the affected node pairs, and $\mathsf{AFFE_m}$ is the affected edges (*i.e.,* those added edges) on G_m after inserting edge (x, y) with $x \in G_m$ and $y \in G_s \cup G_r$. Note that, algorithm incGm2Gsr is bounded by the minimum cover of affected node pairs K_{min} as $\|\mathsf{AFF}\| \leq 2\|K_{min}\|$ and the affected edges $\mathsf{AFFE_m}$.

The time cost arises from (a) the maintenance of partition, (b) the maintenance of local topological order and the SCC detection. (a) It takes $O(|\mathsf{AFFE_m}|)$ time to maintain the partition. Before inserting (x, y), and after handling (x, y) by incGm2Gsr, the partition of G is always valid. All edges starting from y and traversing on $G_s \cup G_r$ are the edges that need to be added to G_m, *i.e.,* $\mathsf{AFFE_m}$. Algorithm incGm2Gsr only traverses on the dummy nodes of $G_s \cup G_r$ in a one-pass manner based on DFS, which traverses at most $|\mathsf{AFFE_m}|$ edges as the inner edges of SCCs in G_s are not traversed. Besides, the number of nodes visited by incGm2Gsr must be less than $|\mathsf{AFFE_m}|$, and thus, it takes $O(|\mathsf{AFFE_m}|)$ time. (b) It also takes $O(|\mathsf{AFFE_m}|)$ time to create new orders when maintaining the local topological orders, except (dx, dy), as each operation of ordered lists takes $O(1)$ time. Besides, incGm2Gm takes $O(\|\mathsf{AFF}\|log\|\mathsf{AFF}\|)$ time to detect the SCCs and maintain the local topological order.

Input: Edge (x, y) with $x, y \in G_{si} \cup G_r$, citation graph G, arrays mSCCs, inGm, inGsi, ordered lists oLists, disjoint set dSet.
Output: Updated G, mSCCs, inGm, inGsi, oLists, dSet.

1. Update G by inserting (x, y);
2. $v_{lo} = +\infty$; $Rdy = \emptyset$; $isExist = false$;
3. $inSCC[dv] = isVisit[dv] = false$ for all $v \in G_s \cup G_r$;
4. **if** $(x, y) \in E_{n2o}$ **then return**
5. **if** $(x, y) \in E_{o2n}$ **then**
6. scanGsGr2(dy, v_{lo});
7. **if** $isExist$ **then**
8. Maintain mSCCs and dSet for SCCs using $inSCC$;
9. Update inGm and maintain the orders using Rdy;
10. **if** $(x, y) \in E_{s2s}$ **then**
11. **if** $x \in G_{si}$, $y \in G_{si}$ **then**
12. Call incGm2Gm for (dx, dy) on G_{si};
13. **if** $x \in G_{si} \cup G_r$, $y \in G_r$ **then**
14. Update inGsi and maintain the order for dy;
15. Repeat line 12 if $x \in G_{si}$;
16. **if** $x \in G_r$, $y \in G_{si}$ **then return**
17. **return** updated G, mSCCs, inGm, inGsi, oLists, dSet.

Figure 10. Algorithm incGsr2Gsr

D. Case $G_s \cup G_r$ to $G_s \cup G_r$

We then present incremental algorithm incGsr2Gsr to handle case $G_s \cup G_r$ to $G_s \cup G_r$ with invalid partition or local topological orders (*i.e.,* Case (4)–(8)) based on the analysis in Section IV.

Algorithm incGm2Gm essentially performs on the DAG representation of G, shown in Fig. 10. For Cases (4) & (8), the partition and local topological order are both valid and nothing needs to be done by Propositions 1, 3 & 7. For Cases (5) & (7), the partitions are invalid, which are handled by the revisions of scanGsGr1 and incGm2Gsr, respectively. For Case (6), the partition is always valid, and the local topological order is invalid which can be maintained by calling incGm2Gm on G_s. The details of incGsr2Gsr is shown below.

Private data structures & variables in incGm2Gsr**.** (1) Arrays Rdy, $isVisit$ and variable v_{lo} are defined the same as incGm2Gsr, and (2) array $inSCC$ indicates whether a dummy node of $G_s \cup G_r$ is in the new SCC or not.

Procedure scanGsGr2 both detects the SCCs and finds all reachable dummy nodes from dy on $G_s \cup G_r$, which is a slight variant of scanGsGr1. Specifically, the following two lines are inserted after line 4 of scanGsGr1: (1) "**if** $dz = dx$ **then** $inSCC[dz] = isExist = true$", and (2) "**if** $inSCC[dy]$ **or** $inSCC[dz]$ **then** $inSCC[dy] = true$". Indeed, it back propagates each member dv of the new SCC with $inSCC[dv] = true$ along the path from dy to dx if exists.

Algorithm incGsr2Gsr takes the same input and output as incGm2Gsr, except that $x, y \in G_{si} \cup G_r$.

(1) It first updates G by inserting (x, y), finds dummy nodes dx, dy, and initializes v_{lo} to $+\infty$, Rdy to empty, $isExist$ to $false$, $isVisit$ and $inSCC$ to $false$ for all dummy nodes of $G_s \cup G_r$ (lines 1-3). (2) If $(x, y) \in E_{n2o}$ (*i.e.,* Case (4)), then nothing needs to be done by Propositions 1, 3 & 7 (line 4). (3) If $(x, y) \in E_{o2n}$ (*i.e.,* Case (5)), it calls scanGsGr2 with dy, v_{lo}, and updates v_{lo}, Rdy, $isVisit$, $isExist$, and $inSCC$. If $isExist = true$, mSCCs and dSet are maintained similar to algorithm incGm2Gm, using $inSCC$ to find all members of the new SCC. Besides, it updates inGm to maintain the

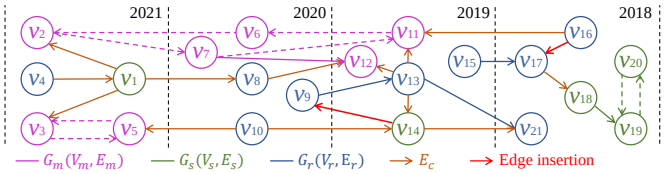


Figure 11. Running example for incGsr2Gsr

partition and creates a decreasing order for Rdy to maintain the local topological order along the same lines as incGm2Gsr (lines 5-9). (4) If $(x, y) \in E_{s2s}$, it further handles Cases (6), (7) & (8) for x and y in G_{si} or G_r (lines 10-16). (a) If $x \in G_{si}$ and $y \in G_{si}$ (*i.e.,* Case (6)), it calls incGm2Gm with dx, dy, G, mSCCs, inGm, inGsi, oLists, dSet only on the dummy nodes of G_{si} to detect the SCCs and maintain the local topological order (lines 11, 12). (b) If $x \in G_{si} \cup G_r$ and $y \in G_r$ (*i.e.,* Case (7)), it only updates inGsi$[y]$ to $y.year$ to maintain the partition by Proposition 4. Then, it creates dy an order lower than its out-neighbors in subgraph G_{si} to maintain the local topological order of dy (lines 13, 14). If $x \in G_{si}$, the invalid edge (dx, dy) *w.r.t.* the local topological order can be easily maintained by repeating line 12 (line 15). (c) If $x \in G_r$ and $y \in G_{si}$ (*i.e.,* Case (8)), it does nothing, similar to Case (4) (line 17). (5) Finally, it returns the updated G, mSCCs, inGm, inGsi, oLists, and dSet (line 16).

We next illustrate algorithm incGsr2Gsr with two examples, as shown in Fig. 11.

Example 6: (1) Consider inserting edge $(v_{14}, v_9) \in E_{o2n}$ with $v_{14} \in G_{s2019}$ and $v_9 \in G_r$ into G along the same drawing style as Example 5. By calling scanGsGr2 to search all reachable dummy nodes from dv_9 on $G_s \cup G_r$, algorithm incGsr2Gsr detects the SCC with $\{dv_9, dv_{13}, dv_{14}\}$, and updates v_{lo} to dv_{11}. Finally, the nodes $\{v_9, v_{13}, v_{14}, v_{21}\}$ are marked in G_m, and two orders lower than dv_{11} are created for dv_{21}, dv_9 such that the topological order of G_m is $ord(dv_3) \prec ord(dv_9) \prec ord(dv_{21}) \prec ord(dv_{11}) \prec ord(dv_{12})$.

(2) Consider inserting $(v_{16}, v_{17}) \in E_{s2s}$ with $v_{16}, v_{17} \in G_r$. Algorithm incGsr2Gsr updates v_{17} in G_{s2018} and creates an order lower than dv_{18}. No SCCs are introduced, and the partition and local topological order of G_{s2018} are valid with $ord(dv_{17}) \prec ord(dv_{18}) \prec ord(dv_{19})$. □

The correctness of incGsr2Gsr is assured as follows.

Theorem 5: *Algorithm* incGsr2Gsr *correctly detects the* SCCs *and maintains the local topological order for case* $G_s \cup G_r$ *to* $G_s \cup G_r$. *Further,* AFF *is bounded,* i.e., $\|\mathsf{AFF}\| \leq 2\|K_{min}\|$, *where* K_{min} *is the minimum cover of affected node pairs.* □

Proof: We prove the correctness of incGsr2Gsr for Cases (4)–(8). For each Case, we show incGsr2Gsr (a) maintains the partition, (b) maintains the local topological order, and (c) detects the SCC if exists. Assume inserting (x, y) with $x, y \in G_s \cup G_r$, and V_{dm}, V_{dsi}, V_{dr} are the sets of dummy nodes of V_m, V_{si} and V_r, respectively.

For Cases (4) & (8), the partition and local topological order are both valid, and no SCC is introduced by Propositions 1, 3 & 7.

For Case (5), (a) the partition of G_m, G_{si} and G_r can be easily maintained with the same analysis as Theorem 4. (b) The topological order of G_s is valid, and only the topological order of G_m needs to be maintained. Actually, there are two types of edges violating the topological order on the DAG representation of G_m, *i.e.,* $E_1 = \{(s,t) | s, t \in V_{dsi} \cup V_{dr}\}$, $E_2 = \{(s,t) | s \in V_{dsi} \cup V_{dr}, t \in V_{dm}\}$. Similar to the analysis of Theorem 4, for each edge $(s,t) \in E_1 \cup E_2$, $ord(s) \prec ord(t)$ always holds as procedure scanGsGr2 is designed by slightly revising procedure scanGsGr1. (c) And the SCC only belongs to subgraph $G_s \cup G_r$ if exists that can be detected by procedure scanGsGr2 on $G_s \cup G_r$.

For Case (6), it is proved by Theorem 3 on G_{si}.

For Case (7), (a) incGsr2Gsr marks y in G_{si}, then the partition is valid based on Proposition 4. The same analysis as Theorem 4, it can be proved that incGsr2Gsr (b) maintains the local topological order, and (c) detects the SCC if exists.

Putting these together, we have proved Theorem 5. □

Time complexity. Algorithm incGsr2Gsr takes $O(\|\mathsf{AFF}\| log\|\mathsf{AFF}\| + |\mathsf{AFFE_m}| + |\mathsf{AFFE_s}|)$ time, where (1) AFF is a cover of the affected node pairs, (2) $\mathsf{AFFE_m}$ is the affected edges (*i.e.,* those added edges) on G_m, and (3) $\mathsf{AFFE_s}$ is the affected edges (*i.e.,* those added edges) on G_s, after inserting edge (x, y) with $x \in G_s \cup G_r$ and $y \in G_s \cup G_r$. Note that incGsr2Gsr is bounded by the minimum cover of affected node pairs K_{min} as $\|\mathsf{AFF}\| \leq 2\|K_{min}\|$, the affected edges $\mathsf{AFFE_m}$ in G_m, and the affected edges $\mathsf{AFFE_s}$ in G_s,

The time cost arises from (a) the maintenance of the partition, (b) the maintenance of the local topological order and the SCC detection.

(a) It takes $O(|\mathsf{AFFE_m}| + |\mathsf{AFFE_{si}}|)$ time to maintain the partition. (i) For Case (5), it takes $O(|\mathsf{AFFE_m}|)$ time using scanGsGr2, with the same analysis as incGm2Gsr. (ii) For Case (7), it takes $O(|\mathsf{AFFE_{si}}|)$ time. Before inserting (x, y), and after handling (x, y) by incGsr2Gsr, the partition of G are both valid. All edges starting from y and reaching to G_{si} are the edges that need to be added to G_{si}, *i.e.,* $\mathsf{AFFE_{si}}$. Algorithm incGm2Gsr only traverses the out-neighbors of dy on G_{si} in a one-pass manner, which actually traverses $|\mathsf{AFFE_{si}}|$ edges and visits less than $|\mathsf{AFFE_{si}}|$ nodes. For the other Cases (4), (6) & (8) of incGsr2Gsr, the partition remains valid.

(b) It takes $O(\|\mathsf{AFF}\| log\|\mathsf{AFF}\| + |\mathsf{AFFE_m}| + |\mathsf{AFFE_{si}}|)$ time to detect the SCCs and maintain the local topological order. (i) For Case (5), it takes $O(|\mathsf{AFFE_m}|)$ time as scanGsGr2 detects the SCC and maintains the local topological order. (ii) For Case (6), incGm2Gm takes $O(\|\mathsf{AFF}\| log\|\mathsf{AFF}\|)$ time on G_{si}. (iii) For Case (7), it takes $O(|\mathsf{AFFE_{si}}| + \|\mathsf{AFF}\| log\|\mathsf{AFF}\|)$ time where traversing the out-neighbors of dy takes $O(|\mathsf{AFFE_{si}}|)$ time, and calling incGm2Gm takes $O(\|\mathsf{AFF}\| log\|\mathsf{AFF}\|)$ time. For the other Cases (4) & (8) of incGsr2Gsr, the local topological order remains valid and no extra SCCs are introduced.

E. The Complete Algorithm

We finally present the complete incremental SCC detection algorithm sinDSCC for single updates by combining the single

node and edge insertions (algorithms incGm2Gm, incGm2Gsr and incGsr2Gsr).

Single update algorithm sinDSCC takes as input a node x or an edge (x, y), citation graph G, arrays mSCCs, inGm, inGsi, ordered lists oLists and disjoint set dSet, and returns the updated G, inGm, inGsi, mSCCs, oLists, dSet. (1) For a single node insertion x, it simply marks the node x in G_r. (2) For a single edge insertion (x, y), it calls algorithms incGm2Gm, incGm2Gsr and incGsr2Gsr for cases G_m to G_m, G_m to $G_s \cup G_r$ and $G_s \cup G_r$ to $G_s \cup G_r$, respectively, and does nothing for case $G_s \cup G_r$ to G_m by Propositions 1, 3 & 7.

Correctness. The correctness of algorithm sinDSCC follows easily from the results of staDSCC based on Propositions 2 & 6 and the correctness of single node and edge insertions by Theorems 3, 4 & 5. Note that the correctness of sinDSCC is independent of the insertion sequence of nodes and edges, as before and after single insertions, the partition and local topological order remain valid.

Time and space complexities. By the time complexity analyses of algorithms incGm2Gm, incGm2Gsr and incGsr2Gsr, it is easy to know that sinDSCC is bounded in $O(\|\mathsf{AFF}\| log \|\mathsf{AFF}\| + |\mathsf{AFFE_m}| + |\mathsf{AFFE_s}|)$ time for single updates. The space complexity of algorithm sinDSCC is $O(|V| + |E|)$.

The space complexity of algorithm sinDSCC is dominated by its key data structures, including citation graph G, shared data structures, *i.e.,* arrays mSCCs, inGm, inGsi, ordered lists oLists, disjoint set dSet, and private data structures, *i.e.,* arrays inF, inB, Rdy, $isVisit$, $inSCC$. The storage of the citation graph costs $O(|V| + |E|)$ space. Each of arrays inGm, inGsi and mSCCs costs $O(|V|)$ space. Each of arrays inF and inB costs at most $O(|V_m|)$ space. Each of arrays Rdy, $isVisit$ and $inSCC$ costs at most $O(|V_s + V_r|)$ space. Ordered lists oLists costs at most $O(|V|)$ space as the number of dummy nodes of G is less than $|V|$.

From these, algorithm sinDSCC takes $O(|V| + |E|)$ space for single updates.

Putting these together, we have proved Theorem 2.

Remarks. Different from (1) the incremental maintenance of (weak) topological order of a DAG [31, 33–37] and (2) incremental SCC detection based on the (weak) topological order for the general graphs [28–30], our sinDSCC detects the SCCs, maintains the partition and local topological order for scholarly data.

VI. Dealing with batch updates

In this section, we present our incremental method to deal with batch updates, which is designed based on our single incremental method sinDSCC, by reducing invalid edges when maintaining the partition and local topological order.

The main result is stated below.

Theorem 6: *Given the original partition and local topological order of citation graph $G(V, E)$ and an increment subgraph $\Delta G(V_\Delta, E_\Delta)$, there exists a bounded incremental algorithm that detects the* SCCs *and maintains the partition and local topological order of $G+\Delta G$ in $O(|\mathsf{AFFE_m}|+|\mathsf{AFFE_s}|+|V_\Delta|+|E_\Delta|\|\mathsf{AFF}\| log \|\mathsf{AFF}\|)$ time, where (1)* AFF *is bounded,* i.e., $\|\mathsf{AFF}\| \leq 2\|K_{min}\|$ *such that K_{min} is the minimum cover of affected node pairs, and (2)* $\mathsf{AFFE_m}$ *and* $\mathsf{AFFE_s}$ *are the affected edges on G_m and on G_s, respectively.* □

Input: Batch updates $\Delta G(V_\Delta, E_\Delta)$, citation graph G, arrays mSCCs, inGm, inGsi, ordered lists oLists, disjoint set dSet.
Output: Updated G, mSCCs, inGm, inGsi, oLists, dSet.

1. Update G by inserting nodes V_Δ;
2. **let** E_{vp} be the valid edges in E_Δ *w.r.t.* the partition;
3. **let** E_{vo} be the valid edges in E_Δ *w.r.t.* the local topological order;
4. Update G by inserting valid edges $E_{vp} \cap E_{vo}$;
5. **let** $E_{invp} = E_\Delta \setminus E_{vp}$;
6. **while** $E_{invp} \neq \emptyset$ **do**
7. Handle the first edge in E_{invp} with sinDSCC;
8. Append newly valid edges to E_{vp} and E_{vo}, respectively;
9. Update G by inserting valid edges $E_{vp} \cap E_{vo}$;
10. $E_{invp} = E_\Delta \setminus E_{vp}$;
11. **let** $E_{invo} = E_\Delta \setminus E_{vo}$;
12. **while** $E_{invo} \neq \emptyset$ **do**
13. Handle the first edge in E_{invo} with sinDSCC;
14. Append newly valid edges to E_{vo};
15. Update G by inserting valid edges E_{vo};
16. $E_{invo} = E_\Delta \setminus E_{vo}$;
17. **return** updated G, mSCCs, inGm, inGsi, oLists, dSet.

Figure 12. Algorithm batDSCC

It is easy to see that algorithm sinDSCC supports continuous updates as the partition and local topological order of the citation graph are always maintained for each update. Hence, for batch updates ΔG, the SCCs can be correctly detected by sinDSCC for each update in ΔG one by one. However, the valid edges *w.r.t.* the partition and local topological order of ΔG, *i.e.,* Cases (3), (4) & (8) are simply inserted to G by algorithm sinDSCC, and we have the following observation.

Heuristic: *For batch updates, reducing invalid edges essentially improves the detecting efficiency.* □

Observe that (1) valid edges *w.r.t.* the partition and local topological order may become invalid when sinDSCC handles invalid edges. (2) When sinDSCC maintains the partition, the invalid edges *w.r.t.* the local topological order may change, *i.e.,* Cases (1) & (6). These tell us that valid edges should be considered first, and invalid edges *w.r.t.* the partition, *i.e.,* Cases (2), (5) & (7)) should be considered before those *w.r.t.* the local topological order. In this way, invalid edges are reduced. This heuristic inspires the design of the batch incremental method.

Algorithm batDSCC takes the same input and output as algorithm sinDSCC shown in Fig. 12, except that it has batch updates $\Delta G(V_\Delta, E_\Delta)$. (1) It first updates G by inserting nodes V_Δ (line 1). (2) Let E_{vp}, E_{vo} be the valid edges in E_Δ *w.r.t.* the partition and local topological order, respectively, and it updates G by inserting valid edges $E_{vp} \cap E_{vo}$ (lines 2-4). (3) The invalid edges E_{invp} *w.r.t.* the partition, *i.e.,* $E_\Delta \setminus E_{vp}$ is processed one by one until E_{invp} is empty (lines 5-10). Note that the invalid edges of E_{invp} in G_m are processed first. (a) It handles the first edge in E_{invp} with sinDSCC, and appends newly valid edges to E_{vp} and E_{vo}, respectively (lines 7, 8). Note that, newly valid edges of E_{vp} can be found by adding flags in procedures scanGsGr1 and scanGsGr2, and newly valid edges of E_{vo} can be found by adding flags in procedure

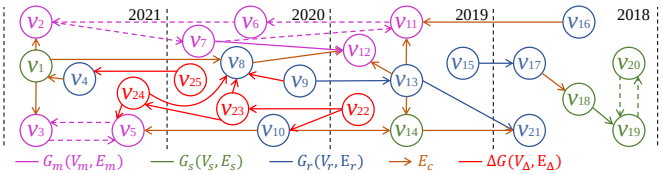


Figure 13. Running example for batch updates

maintain, which does not incur extra time complexities. (b) It then updates G by inserting newly valid edges $E_{vp} \cap E_{vo}$. E_{invp} is updated by $E_\Delta \setminus E_{vp}$ (lines 9, 10). (4) The invalid edges *w.r.t.* the local topological orders are processed similarly (lines 11-16). (5) Finally, it returns the updated G, inGm, inGsi, mSCCs, oLists, and dSet (line 17).

We next illustrate batch updates algorithm with an example, as shown in Fig. 13.

Example 7: Consider inserting ΔG into G along the same drawing style as Example 5, where ΔG has four nodes, *i.e.,* $\{v_{22}, v_{23}, v_{24}, v_{25}\}$, and eight edges, *i.e.,* $\{(v_{23}, v_{24}), (v_9, v_8), (v_{24}, v_5), (v_{22}, v_{10}), (v_{22}, v_{23}), (v_{25}, v_4), (v_{24}, v_8), (v_{23}, v_8)\}$. Based on batDSCC, four nodes are first inserted into G, followed by two valid edges $\{(v_{24}, v_5), (v_{24}, v_8)\}$. The four edges $\{(v_{23}, v_{24}), (v_{22}, v_{10}), (v_{22}, v_{23}), (v_{25}, v_4)\}$ violating the partition are then handled, and each of them traverses the graph once. The two edges $(v_9, v_8), (v_{23}, v_8)$ are updated to valid and inserted without graph traversals. Finally, no extra SCCs are introduced, nodes $\{dv_1, dv_4, dv_8, dv_{10}, dv_{23}, dv_{24}\}$ belong to G_m, and the local topological order is also maintained by batDSCC. Hence, batDSCC only handles 4 invalid edges (visiting 5 nodes, traversing 5 edges), while 8 invalid edges are handled if the edges are inserted one by one following the sequence of ΔG with sinDSCC (visiting 6 nodes, traversing 8 edges). □

Correctness. Algorithm batDSCC is essentially designed based on sinDSCC and utilizes the results of staDSCC by Propositions 2 & 6. Besides, for any sequences of batch updates, the SCCs can be correctly detected by handling updates one by one with sinDSCC. Actually, batDSCC first updates nodes and valid edges $E_{vp} \cap E_{vo}$. It then handles E_{invp} one by one with sinDSCC and updates the newly valid edges $E_{vp} \cap E_{vo}$ until E_{invp} is empty. It also handles E_{invo} one by one with sinDSCC and only updates the newly valid edges E_{vo} until E_{invo} is empty, where the partition is always valid for E_{invo}. That means batch update algorithm batDSCC handles all updates in ΔG one by one with single update algorithm sinDSCC, which assures the correctness.

Time and space complexities. batDSCC takes $O(|\mathsf{AFFE_m}| + |\mathsf{AFFE_s}| + |V_\Delta| + |E_\Delta|\|\mathsf{AFF}\|log\|\mathsf{AFF}\|)$ time for batch updates $\Delta G(V_\Delta, E_\Delta)$, where (1) AFF is bounded, *i.e.,* $\|\mathsf{AFF}\| \leq 2\|K_{min}\|$ such that K_{min} is the minimum cover of affected node pairs, and (2) $\mathsf{AFFE_m}$ and $\mathsf{AFFE_s}$ are the affected edges (*i.e.,* those added edges) on G_m and G_s, respectively.

The time cost of batDSCC arises from (a) node insertions, and (b) edge insertions.

(a) For $|V_\Delta|$ node insertions, it takes $O(|V_\Delta|)$ time.

(b) For $|E_\Delta|$ edge insertions, it takes $O(|\mathsf{AFFE_m}| + |\mathsf{AFFE_s}| + |E_\Delta|\|\mathsf{AFF}\|log\|\mathsf{AFF}\|)$ time. (i) It takes $O(|\mathsf{AFFE_m}| + |\mathsf{AFFE_s}|)$ time to maintain the partition. Algorithm batDSCC first maintains the partition of G_m, which only traverses $|\mathsf{AFFE_m}|$ edges and visits less than $|\mathsf{AFFE_m}|$ nodes in a one-pass manner based on the analysis of sinDSCC. It then maintains the partition of G_s, which traverses $|\mathsf{AFFE_s}|$ edges and visits less than $|\mathsf{AFFE_s}|$ nodes also in a one-pass manner. (ii) It also takes $O(|\mathsf{AFFE_m}| + |\mathsf{AFFE_s}|)$ time to create new orders for the affected nodes (*i.e.,* those added nodes) of G_m and G_s as new orders are obtained when maintaining the partition. (iii) It takes $O(|E_\Delta|\|\mathsf{AFF}\|log\|\mathsf{AFF}\|)$ time to detect the SCCs and maintain the local topological order for all edges of E_Δ. (iv) E_{vp} and E_{vo} can be easily found by adding flags in algorithm sinDSCC, which does not incur extra time complexity.

The space complexity of algorithm batDSCC is $O(|V| + |E| + |V_\Delta| + |E_\Delta|)$, which only introduces extra flags for marking the insert edges valid or invalid based on sinDSCC. The citation graph costs $O(|V| + |E| + |V_\Delta| + |E_\Delta|)$ space, and the space of other data structures are all less than $O(|V| + |V_\Delta|)$.

Putting these together, we have proved Theorem 6.

VII. Experimental Study

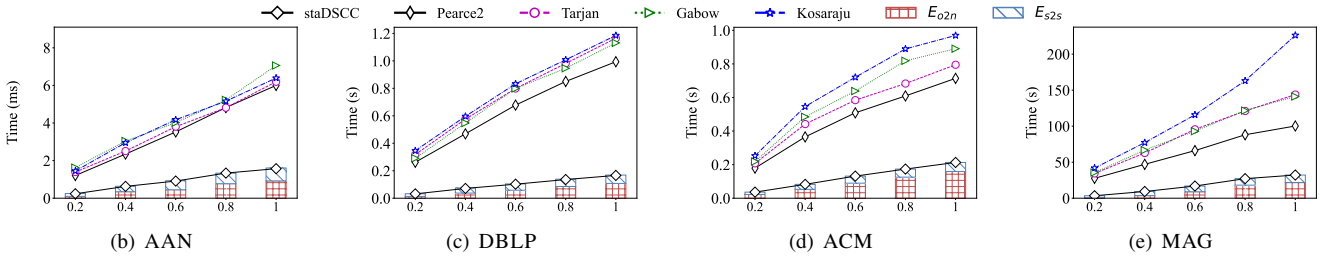
Using four real-life citation graphs, we conduct an extensive experimental study of our static staDSCC, single and batch bounded incremental sinDSCC and batDSCC.

A. Experimental Settings

Datasets. (1) AAN [38] collects the articles published at ACL from 1965 to 2011. (2) DBLP [18] collects the publications at the DBLP Bibliography from 1936 to 2016. (3) ACM [18] also collects in Computer Science from 1936 to 2016. (4) MAG [17] gathers different types of publications *e.g.,* books, conferences, journals and patents of various disciplines from 1800 to 2021. These datasets were further cleaned by deleting the self and redundant citations and the articles with 1000+ references. These datasets are summarized in Table II.

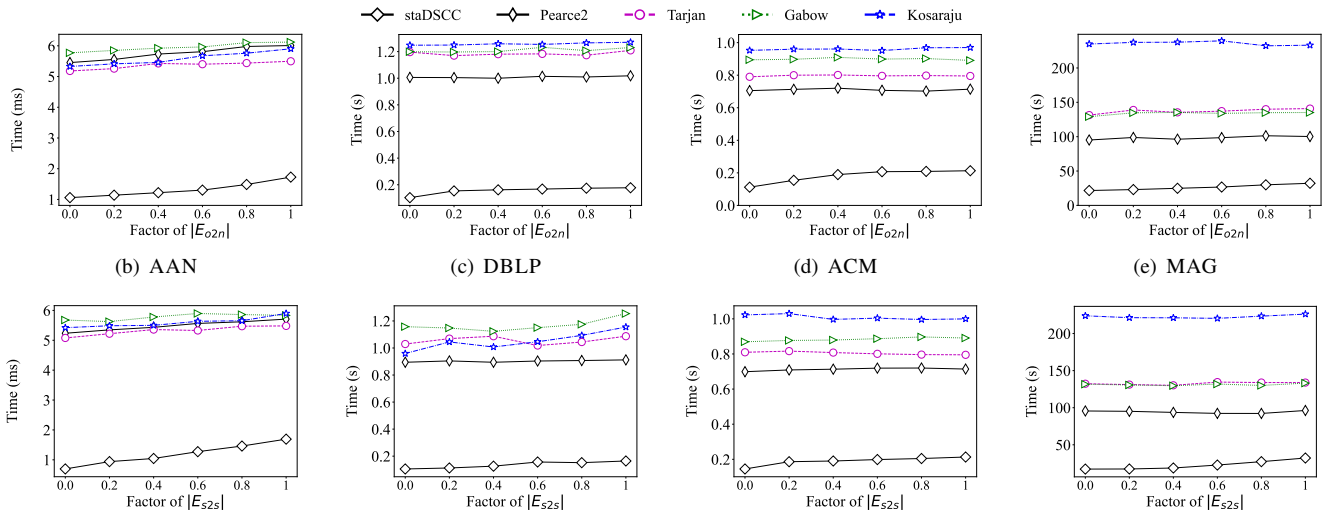
Algorithms. (1) We compared our static staDSCC with four competitive methods: Pearce2 [25], Tarjan [27], Gabow [24] and Kosaraju [26]. (a) Pearce2 is an up-to-date method that reduces the space costs. (b) Tarjan is a classic method that traverses the graph in one-pass. (c) Gabow gives a simplified view of Tarjan without sacrificing efficiency. (d) Kosaraju detects SCCs via DFS on the transpose and original graphs.

(2) We compared our single and batch incremental algorithms sinDSCC and batDSCC with five competitive incremental methods: AHRSZ [31], HKMST [30], incSCC$^+$ [29, 45], MNR [35], PK$_2$ [37]. (a) AHRSZ is a classic method to incrementally maintain the weak topological order of a DAG by iteratively searching from forward and backward. (b) HKMST detects the SCCs and maintains the topological order by using a two-way compatible search. (c) incSCC$^+$ handles batch updates for SCC detection by storing the shared affected areas that are obtained by forward and backward DFS. (d) MNR utilizes forward DFS to find the affected region to support incremental topological ordering. (e) PK$_2$ presents a batch



(b) AAN (c) DBLP (d) ACM (e) MAG

Figure 14. Efficiency tests of static SCC detection



(b) AAN (c) DBLP (d) ACM (e) MAG

(f) AAN (g) DBLP (h) ACM (i) MAG

Figure 15. Efficiency tests of static algorithms: varying E_{o2n} and E_{s2s}

incremental topological ordering based on MNR to reduce redundant graph traversals.

Implementation. All reported time is the execution of static and incremental algorithms that need to perform graph traversals, where the time of graph construction is excluded. When constructing the citation graph, the types of edges E_{o2n}, E_{s2s}, E_{n2o} and max_year, min_year are recorded as the input of our static staDSCC. Besides, incremental algorithms AHRSZ, MNR and PK_2 only maintain the (weak) topological order of a DAG, and terminate once an SCC is detected, we extend them to support continuously detect and maintain SCCs, similar to our batDSCC and staDSCC.

All algorithms were implemented with C++, and graphs were represented by Boost Graph Library [46]. Experiments were conducted on a PC with 2 Intel Xeon E5-2640 2.6GHz CPUs, 128GB RAM, running 64 bit Windows 10 operating system. All tests were repeated over 3 times and the average is reported. Codes of all tested algorithms are available at https://github.com/jfkey/detect-SCC.

B. Experimental Results

We next present our findings.

Exp-1: Tests of static algorithms. In the first set of tests, we compare the running time and space cost of staDSCC with its competitors Pearce2, Tarjan, Gabow and Kosaraju. We also study the impact of $|E_{o2n}|$ and $|E_{s2s}|$ on the efficiency of static algorithms.

Exp-1.1. To evaluate the impacts of the graph size, we vary $|G|$ with the scale factors from 0.2 to 1 that scale both the number of nodes and edges such that $|E|/|V|$ of the scaled citation graph is kept fixed. The results are reported in Fig. 14.

When varying the scale factor, the running time of all algorithms increases linearly. staDSCC consistently runs faster than its competitors, and the running time of Pearce2, Tarjan, Gabow and Kosaraju is close as their time complexities are all $O(|V|+|E|)$. More specifically, staDSCC is (4.0, 6.7, 4.1, 4.7), (4.3, 8.0, 4.8, 6.3), (4.9, 7.7, 4.8, 6.4), (4.7, 8.4, 5.8, 8.0) times faster than Pearce2, Tarjan, Gabow and Kosaraju on (AAN, DBLP, ACM, MAG) on average, respectively. This is consistent with the time complexity analyses, as staDSCC only visits $|V_{ns}|$ nodes and $|E_{ns}|$ edges. Indeed, ($|V_{ns}|$, $|E_{ns}|$) only accounts for (22.1%, 25.8%), (9.7%, 26.6%), (15.1%, 43.5%) and (16.4%, 42.6%) on AAN, DBLP, ACM and MAG, while its static counterparts all scan (100%, 100%) nodes and edges.

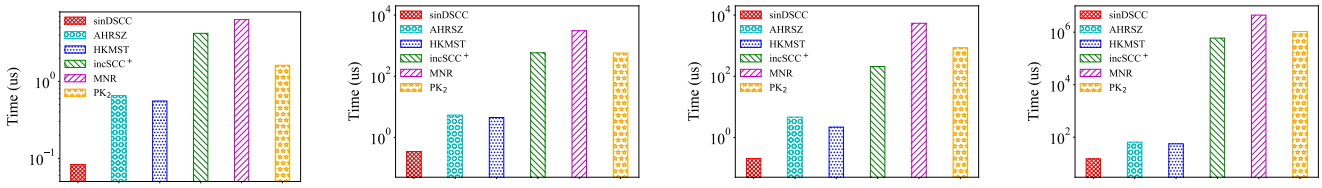
There are in total two cases in our staDSCC to detect the SCCs, *i.e.,* traversing edges in E_{o2n} and E_{s2s}. The time spent by cases traversing edges from the edge heads of E_{o2n} and E_{s2s} of staDSCC all increases with the graph size. Besides, the case of traversing edges in E_{o2n} takes the most time, which occupies (52%, 55%, 68%, 51%) on (AAN, DBLP, ACM, MAG) on average, respectively.

Exp-1.2. To evaluate the space cost, we test the memory cost on the entire datasets. The results are reported in Table III.

The space is mostly consumed by citation graphs, which accounts for (95%, 88%, 92%, 94%) on (AAN, DBLP,

Table III
SPACE COST OF STATIC ALGORITHMS (MB)

Datasets	G	Pearce2	Tarjan	Gabow	Kosaraju	staDSCC
AAN	13.1	0.7	1.0	0.9	0.8	0.3
DBLP	875.7	120.2	132.5	140.5	120.5	24.7
ACM	1031.5	91.1	100.4	105.6	91.3	21.9
MAG	94123.4	7031.9	7656.5	7738.2	6914.4	1145.4



(a) AAN (b) DBLP (c) ACM (d) MAG

Figure 16. Efficiency tests of single update algorithms

Table IV
AVERAGE AFFECTED NODES PER EDGE INSERTION

Datasets	AHRSZ	HKMST	incSCC$^+$	MNR	PK	sinDSCC
AAN	0.17	0.14	13.16	406.35	406.53	0.03
DBLP	0.72	0.55	510.06	115158.77	115176.97	0.04
ACM	0.83	0.63	2045.35	84208.46	84216.70	0.09
MAG	0.22	0.15	17980.82	155719.55	155720.04	0.06

ACM, MAG) on average, respectively. The space cost of extra data structures of staDSCC is $(2.0,\ 4.2, 3.2, 5.4)$ times less than its competitors on (AAN, DBLP, ACM, MAG) on average, respectively. This is because staDSCC follows Pearce2 that reduces the space requirements, and staDSCC avoids unnecessary visits of nodes and edges.

Exp-1.3. To evaluate the impacts of the number of E_{o2n} and E_{s2s}, we separately test the running time of static algorithms when varying $|E_{o2n}|$ and $|E_{s2s}|$. The results are reported in Fig. 15.

To evaluate the impacts of $|E_{o2n}|$, we vary $|E_{o2n}|$ with a factor from 0 to 1 on the entire datasets, where $|E_{s2s}|$ and $|E_{n2o}|$ are kept fixed, *i.e.,* only a factor of edges of E_{o2n} are kept (the other edges of E_{o2n} are randomly deleted), and all the edges of E_{s2s} and E_{n2o} remain untouched in citation graphs. The results are reported in Figs. 15(b) – 15(e).

When varying $|E_{o2n}|$ with a factor from 0 to 1, our static staDSCC consistently runs faster than all static counterparts, and staDSCC is on average (4.2, 6.4, 4.1, 3.8) times faster than the best static competitors on (AAN, DBLP, ACM, MAG), respectively. Besides, when increasing the factor of $|E_{o2n}|$, all static competitors increase slowly due to the low occupancy of E_{o2n} (at most 1.68% in four citation graphs). Our staDSCC also increases slowly due to two reasons. (1) $|E_{o2n}|$ only accounts for less than 1.68% in four citation graphs. (2) Traversing edges in E_{o2n} could find a large of common nodes and edges when traversing edges in E_{s2s}, which leads to the slow increase of $|V_{ns}|$ and $|E_{ns}|$ in staDSCC. Note that it has a time complexity of $\Theta(|V_{ns}| + |E_{ns}|)$.

To evaluate the impacts of $|E_{s2s}|$, we vary $|E_{s2s}|$ with a factor from 0 to 1 on the entire datasets, where $|E_{o2n}|$ and $|E_{n2o}|$ are kept fixed, *i.e.,* only a factor of edges of E_{s2s} are kept (the other edges of E_{s2s} are randomly deleted), and all edges of E_{o2n} and E_{n2o} remain untouched in citation graphs. The results are reported in Figs. 15(f) – 15(i).

When varying $|E_{s2s}|$ with a factor from 0 to 1, our static staDSCC consistently runs faster than all static counterparts, and staDSCC is on average (5.1, 6.8, 3.8, 4.4) times faster than the best static competitors on (AAN, DBLP, ACM, MAG), respectively. Besides, when increasing the factor of $|E_{s2s}|$, our static staDSCC and all its static competitors all increase slowly. The rationale behind this is similar to the case varying $|E_{o2n}|$ with a factor from 0 to 1.

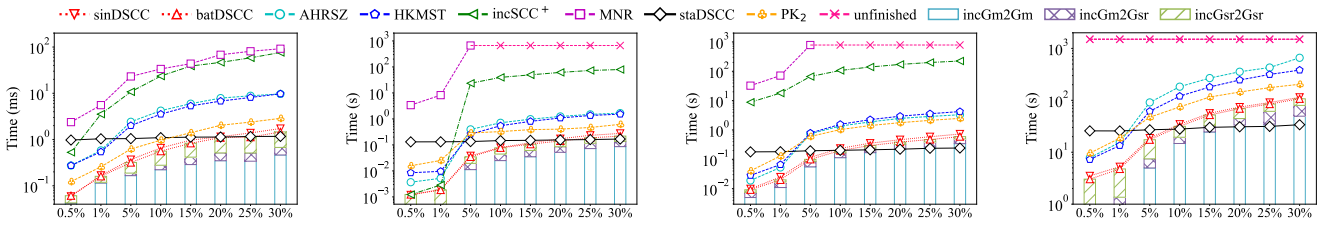
Exp-2: Tests of single update algorithms. In the second set of tests, we compare the running time and the number of affected nodes of single update algorithm sinDSCC with its competitors AHRSZ, HKMST, incSCC$^+$, MNR and PK$_2$. We randomly select $15,000$ edges as inserted edges.

Exp-2.1. To evaluate the efficiency of single updates, we test the running time for inserting $15,000$ edges. The average time per edge insertion is reported in Fig. 16.

Algorithm sinDSCC consistently runs faster than its single update competitors, and is on average (7.9, 15.5, 22.6, 4.3), (6.7, 13.0, 16.9, 3.7), (50.7, 1682.6, 1018.2, 39437.9), (76.4, 8903.0, 26371.1, 295953.2), (19.5, 1658.2, 4147.6, 70562.0) times faster than AHRSZ, HKMST, incSCC$^+$, MNR, PK$_2$ on (AAN, DBLP, ACM, MAG), respectively. The rationale behind this lies in that the affected nodes found by algorithms incSCC$^+$, MNR and PK$_2$ are much larger than sinDSCC, as shown in Table IV and each algorithm requires to maintain the orders for all these affected nodes.

Exp-2.2. To evaluate the number of affected nodes of single updates, we summarize the average affected nodes per edge insertion of incremental algorithms in Table IV. All algorithms require to maintain the orders for their affected nodes.

We find that the average affected nodes per edge insertion found by sinDSCC are on average (8.5, 7.7, 11670.8, 1156576.6, 1156741.1) times less than AHRSZ, HKMST,



(b) AAN (c) DBLP (d) ACM (e) MAG

Figure 17. Efficiency tests of batch update algorithms

Table V
SPACE COST OF INCREMENTAL ALGORITHMS (MB)

Datasets	G	AHRSZ	HKMST	incSCC$^+$	MNR	PK$_2$	batDSCC
AAN	12.2	1.5	1.5	1.4	1.8	1.7	1.7
DBLP	907.5	133.5	145.2	133.0	135.3	131.4	134.0
ACM	958.8	181.4	198.6	180.2	183.3	177.0	186.3
MAG	86971.2	14921.6	15609.4	×	×	15032.2	15595.0

incSCC$^+$, MNR and PK$_2$ on four real-life citation graphs, respectively. Note that, incSCC$^+$, MNR and PK$_2$ are designed based on DFS which leads to traversing more affected nodes.

Exp-3: Tests of batch update algorithms. In the third set of tests, we compare the running time and space cost of batch update algorithm batDSCC with AHRSZ, HKMST, incSCC$^+$, MNR and PK$_2$. We also evaluate the impacts of $|E_{o2n}|$ and $|E_{s2s}|$ on the efficiency and then study the number of edges involved with graph traversals of batch updates. Specifically, we fix G, and vary $|\Delta G|$ from 0.5% to 30% of $|G|$ on each dataset, where the size of ΔG is measured by the number of its edges. Besides, to better simulate real-life updates, half ΔG are selected from the most recent citations, and the other half are randomly selected from the entire citations.

Exp-3.1. To evaluate the impacts of $|\Delta G|$, we vary $|\Delta G|$ from 0.5% to 30% of $|G|$, *i.e.,* from 319 to 19.13K, 33.22K to 1.99M, 23.65K to 1.42M, and 2.81M to 168.65M on AAN, DBLP, ACM and MAG, respectively. The results are reported in Fig. 17, where those methods running more than 6 hours are stopped and marked with pink '×' in dashed lines.

(1) When varying $|\Delta G|$ from 0.5% to 30%, the running time of incremental and static algorithms all increase with $|\Delta G|$. All incremental algorithms are sensitive to $|\Delta G|$, while staDSCC increases slowly. Our static staDSCC performs better when $|\Delta G|$ is large. The intersection points of best batch incremental competitors and staDSCC are about $(12\%, 4\%, 3\%, 2\%)$ on (AAN, DBLP, ACM, MAG), respectively. However, the intersection points of batDSCC and staDSCC are about $(25\%, 22\%, 10\%, 10\%)$ on (AAN, DBLP, ACM, MAG), respectively, which are much larger than its best competitors. Algorithm batDSCC is on average $(4.5, 8.1, 30.1, 5.2)$ times faster than algorithm staDSCC on (AAN, DBLP, ACM, MAG), respectively.

Note that batDSCC and all known incremental algorithms are not consistently faster than static algorithms [28–31]. Based on its time complexity $O(|\mathsf{AFFE_m}|+|\mathsf{AFFE_s}|+|V_\Delta|+|E_\Delta|\|\mathsf{AFF}\|log\|\mathsf{AFF}\|)$, where $|\mathsf{AFFE_m}|+|\mathsf{AFFE_s}|+|V_\Delta|$ is consistently less than $|V_{ns}|+|E_{ns}|$. The choice between batDSCC and staDSCC mainly depends on $\|\mathsf{AFF}\|$ and $|E_\Delta|$ whether satisfies $c|E_\Delta|\|\mathsf{AFF}\|log\|\mathsf{AFF}\|$ is less than $|V_{ns}|+|E_{ns}|$, where c is a const factor related to the complexity.

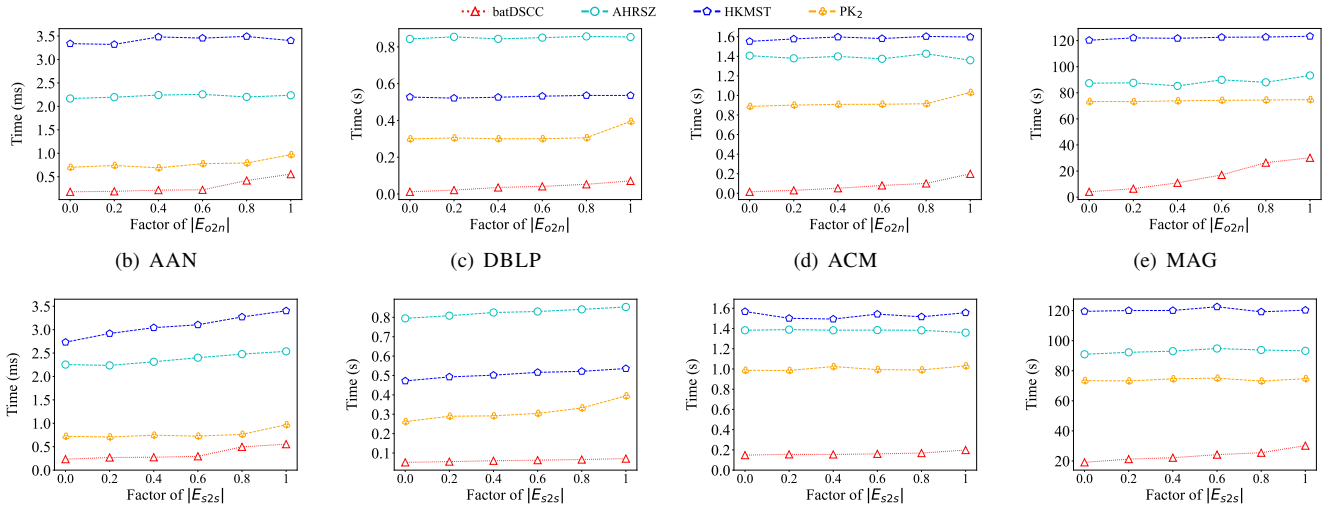
(2) Algorithm batDSCC consistently runs faster than its incremental counterparts before the intersection points. Specifically, batDSCC is on average (6.5, 7.8, 4.8, 4.4), (5.8, 5.1, 5.6, 3.4), (35.4, 348.6, 745.1, 5000+), (51.6, 5000+, 4801.5, 5000+), (1.9, 9.0, 5.6, 3.2) times faster than AHRSZ, HKMST, incSCC$^+$, MNR, PK$_2$ on (AAN, DBLP, ACM, MAG), respectively. This is because (a) algorithm batDSCC does not require to traverse the graph for all inserted edges in G_r and E_c, (b) batDSCC only focuses on subgraphs G_m and G_s, which finds a smaller cover of affected node pairs, and (c) batDSCC further reduces invalid edges for batch updates.

When varying $|\Delta G|$ from 0.5% to 30%, algorithms incGm2Gm, incGm2Gsr and incGsr2Gsr in batDSCC all increase with the increment of $|\Delta G|$. Algorithms incGsr2Gsr and incGm2Gm take the most and second most time, which occupy (45%, 53%, 56%, 48%) and (45%, 27%, 17%, 24%) on (AAN, DBLP, ACM, MAG) on average, respectively.

(3) Batch updates can be handled by sinDSCC for each update in $|\Delta G|$ one by one, *i.e.,* batch version sinDSCC. When varying $|\Delta G|$ from 0.5% to 30%, batDSCC consistently runs faster than this batch version sinDSCC, and reduces $(10.4\%, 10.2\%, 20.5\%, 9.5\%)$ running time on (AAN, DBLP, ACM, MAG) on average, respectively. The improvement is consistent with the analyses of batch updates in Section VI. Note that this is particularly useful when the citation graphs update frequently.

Exp-3.2. To evaluate the space cost of incremental algorithms, we test the memory cost in practice when fixing $|\Delta G|$ = 30%. The space cost analyses include the cost of citation graphs and data structures, and the results are reported in Table V.

Similar to static algorithms, most memory is consumed by the citation graphs, which accounts for $(88\%, 87\%, 84\%, 85\%)$ on (AAN, DBLP, ACM, MAG) on average, respectively. Besides, the data structures space cost of batDSCC and its incremental counterparts are very close, as their data structures all take $O(|V|+|V_\Delta|)$ space. This is consistent with the space complexity analysis of batDSCC.



(b) AAN (c) DBLP (d) ACM (e) MAG

(f) AAN (g) DBLP (h) ACM (i) MAG

Figure 18. Efficiency tests of incremental algorithms: varying E_{o2n} and E_{s2s}

Table VI
THE NUMBER OF EDGES PERFORMING GRAPH TRAVERSALS OF BATCH UPDATES

Datasets	Number of edge insertions	AHRSZ	HKMST	incSCC$^+$	MNR	PK_2	sinDSCC	batDSCC
		Number of invalid weak TO edges	Number of invalid TO edges				Number of invalid partition or local TO edges	
AAN	19,131	1,768	1,698	1,648	1,858	2,148	465	442
DBLP	1,419,597	229,722	229,786	191,416	200,388	582,689	39,441	30,473
ACM	1,995,123	305,341	302,749	235,545	258,084	788,057	64,837	48,192
MAG	168,715,717	13,593,749	11,689,496	×	×	12,455,370	1,850,840	1,817,776

TO refers to topological order.

Exp-3.3. To evaluate the impacts of the number of E_{o2n} and E_{s2s}, we separately test the running time of incremental algorithms when varying $|E_{o2n}|$ and $|E_{s2s}|$. The results are reported in Fig. 18.

To evaluate the impacts of $|E_{o2n}|$, we vary $|E_{o2n}|$ with a factor from 0 to 1 on both G and ΔG of (AAN, DBLP, ACM, MAG), where $|E_{s2s}|$ and $|E_{n2o}|$ are kept fixed and $|\Delta G| = 10\%$, *i.e.,* only a factor of edges of E_{o2n} are kept (the other edges of E_{o2n} are randomly deleted) and all the edges of E_{s2s} and E_{n2o} remain untouched in citation graphs. The results are reported in Figs. 18(b) – 18(e).

When varying $|E_{o2n}|$ with a factor from 0 to 1, our incremental batDSCC consistently runs faster than all incremental counterparts, and batDSCC is on average (2.8, 11.1, 21.9, 7.6) times faster than the best incremental competitors on (AAN, DBLP, ACM, MAG), respectively. Besides, when increasing the factor of $|E_{o2n}|$, all incremental algorithms increase slowly due to the low occupancy of E_{o2n}. The increase of our batDSCC is caused by the following reason. When increasing the factor of $|E_{o2n}|$, subgraph G_m becomes much larger, which generally leads to the significant increase of $\|\mathsf{AFF}\|$ and $|\mathsf{AFFE_m}|$. Note that it has a time complexity of $O(|\mathsf{AFFE_m}| + |\mathsf{AFFE_s}| + |V_\Delta| + |E_\Delta|\|\mathsf{AFF}\|log\|\mathsf{AFF}\|)$.

To evaluate the impacts of $|E_{s2s}|$, we vary $|E_{s2s}|$ with a factor from 0 to 1 on both G and ΔG of (AAN, DBLP, ACM, MAG), where $|E_{o2n}|$ and $|E_{n2o}|$ are kept fixed and $|\Delta G| = 10\%$, *i.e.,* only a factor of edges of E_{s2s} are kept (the other edges of E_{s2s} are randomly deleted) and all the edges of E_{o2n} and E_{n2o} remain untouched in citation graphs. The results are reported in Figs. 18(f) – 18(i).

When varying $|E_{s2s}|$ with a factor from 0 to 1, our incremental batDSCC consistently runs faster than all incremental counterparts, and batDSCC is on average (2.5, 5.1, 5.6, 3.2) times faster than the best incremental competitors on (AAN, DBLP, ACM, MAG), respectively. Besides, when increasing the factor of $|E_{s2s}|$, all incremental algorithms increase slowly due to the low occupancy of E_{s2s}. The increase of our batDSCC is caused by the following two reasons. (1) $|E_{s2s}|$ only accounts for a low ratio. (2) When increasing the factor of $|E_{s2s}|$, subgraph G_m is kept fixed and subgraph G_s increases slowly, as it only increases the nodes and edges of G_r and E_c, which leads to the slow increase of $|\mathsf{AFFE_s}|$ and $\|\mathsf{AFF}\|$.

Exp-3.4. To evaluate the number of edges performing graph traversals, we summarize the total number of edges performing graph traversals for batch updates $|\Delta G| = 30\%$ in Table VI. Note that, AHRSZ is designed based on the weak topological order, and HKMST, incSCC$^+$, MNR, PK_2 are all designed based on the topological order. sinDSCC and batDSCC are based on the partition and local topological order. Algorithms perform graph traversals for edges with invalid (weak) topological order or invalid partition and local topological order.

We find the following. First, sinDSCC and batDSCC both reduce at least (87%, 94%, 92%, 95%) invalid edges than its best incremental competitors on (AAN, DBLP, ACM, MAG), respectively. Second, batDSCC further reduces (5%, 23%, 26%, 2%) invalid edges than sinDSCC on (AAN,

DBLP, ACM, MAG), respectively. Third, once incremental SCC detection for scholarly data is designed based on topological order or its variants, most inserted edges ($> 83.82\%$) are valid, *i.e.,* the graph traversal is unnecessary. To conclude, our incremental algorithms sinDSCC and batDSCC capture the properties of citation graphs, and batDSCC further reduces the amount of graph traversals, which is consistent with the analyses in Sections IV, V & VI.

Exp-4: The impact of SCCs on PageRank. In the fourth set of tests, we study the impact of SCCs on computing PageRank scores, as the detected SCCs can be utilized to clean incorrect citations to improve the quality of the scholarly data.

To evaluate the impact of SCCs on PageRank scores, for each SCC, we randomly delete at most ten edges of E_{o2n} and E_{s2s} from the SCC each time until only singleton SCCs remain. We compare the gap between the PageRank scores of the nodes before and after removing all SCCs. The PageRank gap of node i is defined as $|p_i - p'_i|/p'_i$, where p_i and p'_i are the PageRank scores of node i before and after edge deletions, respectively. Besides, we follow [6] to compute the PageRank scores of citation graphs, where each SCC is treated as a block in the process [6, 19]. For the PageRank algorithm of [6], the time decaying factor, damping parameter and iteration threshold are set to -1, 0.85 and 10^{-8}, respectively. The results are reported in Table VII.

After all SCCs are removed, in total of (0.5%, 0.3%, 0.3%, 0.04%) the edges are deleted from (AAN, DBLP, ACM, MAG), respectively. The average PageRank gaps of all nodes on the four citation graphs are over 1.1%. Besides, a more significant gap of the nodes in SCCs is introduced, which achieves (13.2%, 11.9%, 9.4%, 11.1%) on (AAN, DBLP, ACM, MAG), respectively. That is, the SCCs have significant impacts on computing PageRank scores.

Summary. We have the following findings from these tests. (1) Our static algorithm staDSCC is both time and space efficient. Indeed, staDSCC is $(4.9, 5.9, 6.0, 6.7)$ times faster and uses $(2.0, 4.2, 3.2, 5.4)$ times less space than (Pearce2, Tarjan, Gabow and Kosaraju) on average, respectively. (2) The intersection points of batDSCC and staDSCC are about $(25\%, 22\%, 15\%, 10\%)$ for (AAN, DBLP, ACM, MAG), respectively. Before the intersection points, batDSCC is on average $(5.8, 5.0, 35.4+, 51.6+, 5.0)$ times faster than (AHRSZ, HKMST, incSCC$^+$, MNR and PK$_2$), respectively. (3) Most of the memory is consumed by citation graphs, generally more than 80%, and the data structure space cost of incremental algorithms is close. (4) The SCCs have significant impacts on the PageRank scores, and the average gaps are (2.9%, 1.1%, 1.3%, 1.1%) on (AAN, DBLP, ACM, MAG) after removing SCCs, respectively.

Table VII
PAGERANK SCORE GAP AFTER REMOVING SCCS

Datasets	AAN	DBLP	ACM	MAG
Average gap of all nodes	2.9%	1.1%	1.3%	1.1%
Average gap of the nodes in SCCs	13.2%	11.9%	9.4%	11.1%

VIII. RELATED WORK

Static SCC detection. Static SCC detection methods [24–27] compute the SCCs for a given graph $G(V, E)$, which are also known as batch methods. Existing static methods all traverse the entire graph in a one or two pass manner, and take $O(|V|+|E|)$ time. Tarjan [27] employs the property that nodes of an SCC form a subtree in the spanning tree of the graph, and detects SCCs via repeated DFS over the entire graph in a one-pass manner. Based on the idea of Tarjan, Gabow [24] and Pearce2 [25] present one-pass algorithms to reduce the space requirements. Moreover, Kosaraju [26] employs the property that the transpose and original graph share the same SCCs, and it detects SCCs in two-pass via DFS on the two graphs. Differently, our static algorithm divides edges into three types and exploits the properties of different types of edges to reduce the traversal of unnecessary nodes and edges.

Incremental SCC detection. Graphs are naturally dynamic and continuously growing, and incremental SCC detection methods [28–31, 33–37, 45] have been extensively studied.

(1) Algorithms in [31, 33–37] incrementally maintain the (weak) topological order of a DAG only, and terminate once an SCC is detected. AHRSZ [31], MNR [35], PK [36] and PK$_2$ [37] can be extended to support continuously detect and maintain SCCs, similar to our staDSCC and batDSCC. However, BC [33] and BK [34] use a different concept of node equivalence to maintain the topological order, which can not be extended. (2) Algorithms in [28–30, 45] detect and maintain SCCs for general graphs. BFGT [28] and HKMST [30] take $O(|E|^{3/2})$ time for inserting $|E|$ edges one by one into graphs with $|V|$ nodes. incSCC$^+$[29, 45] presents a bounded relative to a static algorithm to handle both single and batch updates, and defines an affected area based on Tarjan [27]. Further, BFGT [28] depends on the level status of each node generated during inserting edges, and can hardly be used to incrementally detect SCCs as they do not support the batch update of the level status.

Different from incremental methods in (1) [31, 33–37] and (2) [28–30] for general graph, our batDSCC takes $O(|\mathsf{AFFE_m}| + |\mathsf{AFFE_s}| + |V_\Delta| + |E_\Delta|\|\mathsf{AFF}\|log\|\mathsf{AFF}\|)$ time for batch updates $\Delta G(V_\Delta, E_\Delta)$ on citation graphs, which is bounded by the minimum cover of affected node pairs K_{min} and the affected edges $\mathsf{AFFE_m}$ and $\mathsf{AFFE_s}$. Besides, the factor $O(|\mathsf{AFFE_m}| + |\mathsf{AFFE_s}|)$ in ours arises from the maintenance of the partition, which is not considered for these algorithms. Specifically, (1) modified AHRSZ, MNR, PK and PK$_2$ can detect and maintain SCCs. AHRSZ takes $O(|E_\Delta|\|\mathsf{AFF}\|log\|\mathsf{AFF}\|)$ time, and its $\|\mathsf{AFF}\|$ is larger than ours, as it is defined on G, instead of subgraphs. Further, the covers of affected node pairs in MNR, PK and PK$_2$ are also larger than our $\|\mathsf{AFF}\|$ [36] as they aim for simple solutions. (2) HKMST and BFGT are bounded by $|E|$, which fail to capture the least amount of nodes to be visited when detecting SCCs, and their time complexities are incomparable with ours. incSCC$^+$ [29, 45] handles batch updates by storing the shared

affected areas to process intra-component and inter-component updates, while batDSCC directly reduces the graph traversal using a heuristic tailored to citation graph updates. Besides, the affected areas of incSCC$^+$ are bounded by Tarjan (using DFS for invalid topological orders), which is the cover of affected node pairs of edge insertion, rather than the minimum cover in this study.

Algorithm incGm2Gm in our incremental methods is inspired by AHRSZ [31]. Differently, (1) incGm2Gm defines a cover of affected node pairs of edge insertions, and modifying procedure discover to find the cover bounded by $\|K_{min}\|$, (2) incGm2Gm only performs on the dummy nodes of subgraph G_m, while AHRSZ performs on the entire graphs and some unnecessary nodes and edges are visited, and (3) procedure maintain is simplified by removing the floor computation ($O(\|\mathsf{AFF}\|log\|\mathsf{AFF}\|)$ [31, 36]), and the number of newly created orders is relaxed to keep the total order of the nodes.

In short, SCC detection is a fundamental graph analytic problem on citation graphs that plays a significant role in scholarly data analyses. Different from existing detection methods for general graphs, our study focuses on detecting SCCs in citation graphs, by exploiting the properties of scholarly data.

Conclusions

We have proposed static algorithm staDSCC to detect SCCs in citation graphs, and bounded single and batch update algorithms sinDSCC and batDSCC to incrementally detect SCCs and support continuous updates of citation graphs. The design of our algorithms is based on the analyses of citation graphs, graph partitioning and local topological order. Finally, we have conducted extensive experiments to verify the efficiency of algorithms staDSCC, batDSCC and sinDSCC on four real-life citation graphs.

A couple of topics need a further study. One is to handle deletions for scholarly data updates although this is a rare case [2, 29, 41, 47, 48], and the other is to clean incorrect citations based on detected SCCs to improve the quality of the scholarly data analytic tasks [21, 22, 49].

References

[1] J. Liu, S. Ma, R. Hu, C. Hu, and J. Huai, "Athena: A ranking enabled scholarly search system," in *WSDM*, 2020.

[2] J. Tang, J. Zhang, L. Yao, J. Li, L. Zhang, and Z. Su, "Arnetminer: extraction and mining of academic social networks," in *SIGKDD*, 2008.

[3] J. Wu, K. M. Williams, H. Chen, M. Khabsa, C. Caragea, S. Tuarob, A. Ororbia, D. Jordan, P. Mitra, and C. L. Giles, "Citeseerx: AI in a digital library search engine," *AI Mag.*, vol. 36, no. 3, pp. 35–48, 2015.

[4] I. Kanellos, T. Vergoulis, D. Sacharidis, T. Dalamagas, and Y. Vassiliou, "Impact-based ranking of scientific publications: A survey and experimental evaluation," *IEEE Trans. Knowl. Data Eng.*, vol. 33, no. 4, pp. 1567–1584, 2021.

[5] T. V. Ilias Kanellos, D. Sacharidis, T. Dalamagas, and Y. Vassiliou, "Ranking papers by their short-term scientific impact," in *ICDE*, 2021.

[6] S. Ma, C. Gong, R. Hu, D. Luo, C. Hu, and J. Huai, "Query independent scholarly article ranking," in *ICDE*, 2018.

[7] Z. Chen, A. Sun, and X. Xiao, "Incremental community detection on large complex attributed network," *ACM Trans. Knowl. Discov. Data*, vol. 15, no. 6, pp. 1–20, 2021.

[8] C. Giatsidis, D. M. Thilikos, and M. Vazirgiannis, "D-cores: measuring collaboration of directed graphs based on degeneracy," *Knowl. Inf. Syst.*, vol. 35, no. 2, pp. 311–343, 2013.

[9] J. Kim and J. Lee, "Community detection in multi-layer graphs: A survey," *SIGMOD Rec.*, vol. 44, no. 3, pp. 37–48, 2015.

[10] D. Luo, S. Ma, Y. Yan, C. Hu, X. Zhang, and J. Huai, "A collective approach to scholar name disambiguation," *IEEE Trans. Knowl. Data Eng.*, vol. 34, no. 5, pp. 2020–2032, 2022.

[11] K. M. Pooja, S. Mondal, and J. Chandra, "Exploiting higher order multi-dimensional relationships with self-attention for author name disambiguation," *ACM Trans. Knowl. Discov. Data*, vol. 16, no. 5, pp. 88:1–88:23, 2022.

[12] J. Tang, A. C. M. Fong, B. Wang, and J. Zhang, "A unified probabilistic framework for name disambiguation in digital library," *IEEE Trans. Knowl. Data Eng.*, vol. 24, no. 6, pp. 975–987, 2012.

[13] M. Burch, K. B. Ten Brinke, A. Castella, G. K. S. Peters, V. Shteriyanov, and R. Vlasvinkel, "Dynamic graph exploration by interactively linked node-link diagrams and matrix visualizations," *Visual Computing for Industry, Biomedicine, and Art*, vol. 4, no. 1, pp. 1–14, 2021.

[14] P. Federico, F. Heimerl, S. Koch, and S. Miksch, "A survey on visual approaches for analyzing scientific literature and patents," *IEEE Trans. Vis. Comput. Graph.*, vol. 23, no. 9, pp. 2179–2198, 2017.

[15] Z. Guo, J. Tao, S. Chen, N. V. Chawla, and C. Wang, "SD2: slicing and dicing scholarly data for interactive evaluation of academic performance," *CoRR*, vol. abs/2203.12671, 2022.

[16] K. Li, H. Yang, E. Montoya, A. Upadhayay, Z. Zhou, J. Saad-Falcon, and D. H. Chau, "Visual exploration of literature with argo scholar," in *CIKM*, 2022.

[17] A. Sinha, Z. Shen, Y. Song, H. Ma, D. Eide, B. P. Hsu, and K. Wang, "An overview of microsoft academic service (MAS) and applications," in *WWW*, 2015.

[18] "Aminer," https://www.aminer.cn/citation, 2022.

[19] P. Berkhin, "Survey: A survey on pagerank computing," *Internet Math.*, vol. 2, no. 1, pp. 73–120, 2005.

[20] S. Li, Z. Zhou, A. Upadhayay, O. Shaikh, S. Freitas, H. Park, Z. J. Wang, S. Routray, M. Hull, and D. H. Chau, "Argo lite: Open-source interactive graph exploration and visualization in browsers," in *CIKM*, 2020.

[21] J. Sun, D. Ajwani, P. K. Nicholson, A. Sala, and S. Parthasarathy, "Breaking cycles in noisy hierarchies," in *WebSci*, 2017, pp. 151–160.

[22] Y. Peng, X. Lin, M. Yu, W. Zhang, and L. Qin, "Tdb: Breaking all hop-constrained cycles in billion-scale directed graphs," *arXiv preprint arXiv:2209.05909*, 2022.

[23] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[24] H. N. Gabow, "Path-based depth-first search for strong and biconnected components," *Inf. Process. Lett.*, vol. 74, no. 3-4, pp. 107–114, 2000.

[25] D. J. Pearce, "A space-efficient algorithm for finding strongly connected components," *Inf. Process. Lett.*, vol. 116, no. 1, pp. 47–52, 2016.

[26] M. Sharir, "A strong-connectivity algorithm and its applications in data flow analysis," *Computers & Mathematics with Applications*, vol. 7, no. 1, pp. 67–72, 1981.

[27] R. E. Tarjan, "Depth-first search and linear graph algorithms," *SIAM J. Comput.*, vol. 1, no. 2, pp. 146–160, 1972.

[28] M. A. Bender, J. T. Fineman, S. Gilbert, and R. E. Tarjan, "A new approach to incremental cycle detection and related problems," *ACM Trans. Algorithms*, vol. 12, no. 2, pp. 14:1–14:22, 2016.

[29] W. Fan, C. Hu, and C. Tian, "Incremental graph computations: Doable and undoable," in *SIGMOD*, 2017.

[30] B. Haeupler, T. Kavitha, R. Mathew, S. Sen, and R. E. Tarjan, "Incremental cycle detection, topological ordering, and strong component maintenance," *ACM Trans. Algorithms*, vol. 8, no. 1, pp. 3:1–3:33, 2012.

[31] B. Alpern, R. Hoover, B. K. Rosen, P. F. Sweeney, and F. K. Zadeck, "Incremental evaluation of computational circuits," in *SODA*, 1990.

[32] M. A. Bender, J. T. Fineman, and S. Gilbert, "A new approach to incremental topological ordering," in *SODA*, 2009.

[33] A. Bernstein and S. Chechik, "Incremental topological sort and cycle detection in expected total time," in *SODA*, 2018.

[34] S. Bhattacharya and J. Kulkarni, "An improved algorithm for incremental cycle detection and topological ordering in sparse graphs," in *SODA*, 2020.

[35] A. Marchetti-Spaccamela, U. Nanni, and H. Rohnert, "Maintaining a topological order under edge insertions," *Inf. Process. Lett.*, vol. 59, no. 1, pp. 53–58, 1996.

[36] D. J. Pearce and P. H. J. Kelly, "A dynamic topological sort algorithm for directed acyclic graphs," *ACM J. Exp. Algorithmics*, vol. 11, pp. 1–7, 2006.

[37] David J. Pearce and Paul H. J. Kelly, "A batch algorithm for maintaining a topological order," in *ACSC*, 2010.

[38] D. R. Radev, P. Muthukrishnan, V. Qazvinian, and A. Abu-Jbara, "The ACL anthology network corpus," *Lang. Resour. Evaluation*, vol. 47, no. 4, pp. 919–944, 2013.

[39] M. A. Bender, J. T. Fineman, S. Gilbert, T. Kopelowitz, and P. Montes, "File maintenance: When in doubt, change the layout!" in *SODA*, 2017.

[40] P. F. Dietz and D. D. Sleator, "Two algorithms for maintaining order in a list," in *STOC*, 1987.

[41] Z. Wu, J. Wu *et al.*, "Towards building a scholarly big data platform: Challenges, lessons and opportunities," in *JCDL*, 2014.

[42] C. Li, J. Han, G. He, X. Jin, Y. Sun, Y. Yu, and T. Wu, "Fast computation of simrank for static and dynamic information networks," in *EDBT*, 2010.

[43] W. Fan, R. Jin, M. Liu, P. Lu, X. Luo, R. Xu, Q. Yin, W. Yu, and J. Zhou, "Application driven graph partitioning," in *SIGMOD*, 2020.

[44] I. Katriel and H. L. Bodlaender, "Online topological ordering," *ACM Trans. Algorithms*, vol. 2, no. 3, pp. 364–379, 2006.

[45] W. Fan and C. Tian, "Incremental graph computations: Doable and undoable," *ACM Trans. Database Syst.*, vol. 47, no. 2, pp. 6:1–6:44, 2022.

[46] J. Siek, L.-Q. Lee, A. Lumsdaine *et al.*, *The boost graph library*. Pearson India, 2002.

[47] A. Bernstein, M. Probst, and C. Wulff-Nilsen, "Decremental strongly-connected components and single-source reachability in near-linear time," in *STOC*, 2019.

[48] L. Roditty and U. Zwick, "Improved dynamic reachability algorithms for directed graphs," *SIAM J. Comput.*, vol. 37, no. 5, pp. 1455–1471, 2008.

[49] P. Li, X. Rao, J. Blase, Y. Zhang, X. Chu, and C. Zhang, "Cleanml: A study for evaluating the impact of data cleaning on ML classification tasks," in *ICDE*, 2021.