Incremental Detection of 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. To fill in this gap, we focus on incrementally detecting strongly connected components (SCCs) for citation graphs of scholarly data in this study. (1) We first analyze how to partition the citation graph and maintain the local topological order to reduce the traversal of unnecessary nodes and edges, by exploiting the properties of different edge types. (2) Based on the analyses, we design an efficient bounded incremental method to handle continuous single updates by dynamically maintaining the partition and local topological order. (3) We then design an efficient bounded batch incremental method by reducing the traversal of unnecessary edges, based on our single update method, to further improve the efficiency for continuous batch updates. (4) Finally, we conduct an experimental study to verify the efficiency of our incremental methods for citation graphs.**

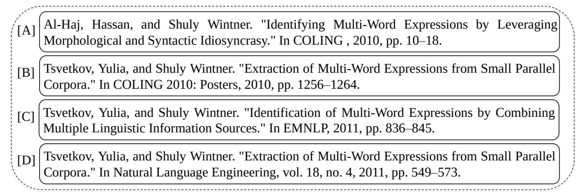
***Index Terms*—Incremental algorithms; SCC detection; Scholarly data analysis**

I. Introduction

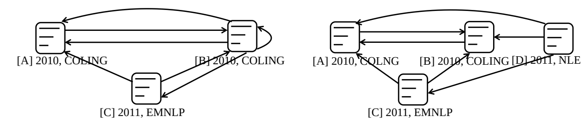
In recent years, the rapid development of science and technology comes with the prosperity of scientific publications, and scholarly data has drawn significant attention from both industrial and academic communities. Analyses of scholarly data help to understand evolving scientific trends and assist scholars to find needed information timely and conveniently [1–3], among which citation graph 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) commonly exist in citation graphs of scholarly data, 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 the same article. We illustrate these with a real-life SCC example shown below.

Example 1: Fig. 1(b) depicts an SCC in the citation graph of DBLP [18] formed due to the two reasons, where nodes A, B, C and D represent the articles shown in Fig. 1(a). (1) The edges (A, B) and (B, A) are formed by the *time gap*. That is, after articles A and B are accepted, the authors add each other to their references as they study the same topic, *i.e.,* mutual citations. (2) The edges (B, A), (B, C) and (B, B) are caused by *wrongly* treating another article D as B. These lead


[A] Al-Haj, Hassan, and Shuly Wintner. "Identifying Multi-Word Expressions by Leveraging Morphological and Syntactic Idiosyncrasy." In COLING , 2010, pp. 10–18.
[B] Tsvetkov, Yulia, and Shuly Wintner. "Extraction of Multi-Word Expressions from Small Parallel Corpora." In COLING 2010: Posters, 2010, pp. 1256–1264.
[C] Tsvetkov, Yulia, and Shuly Wintner. "Identification of Multi-Word Expressions by Combining Multiple Linguistic Information Sources." In EMNLP, 2011, pp. 836–845.
[D] Tsvetkov, Yulia, and Shuly Wintner. "Extraction of Multi-Word Expressions from Small Parallel Corpora." In Natural Language Engineering, vol. 18, no. 4, 2011, pp. 549–573.

(a) Example articles in DBLP [18]



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

Figure 1. A real-life SCC formed by the two reasons

to the formation of the SCC with the nodes {A, B, C}. Indeed, article D has the same title and authors as B, and is the journal version of B. In fact, it is D cites A, B and C, where the true citation graph is shown in Fig. 1(c). □

The dynamic nature of scholarly data and its various applications highlight the necessity for incremental SCC detection on citation graphs. (1) Citation graphs are typically large and constantly growing with relatively 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, 19], and it is impractical to recompute the SCCs from scratch once it gets updated. (2) The incremental SCC detection can be used to support incremental block-wise PageRank computation for article ranking [6], where SCCs are detected first and each SCC is treated as a block in the process [6, 20]. Besides, the detected SCCs provide views and perspectives for visualizing the citation graph topology [13], where the incremental visual exploration is supported with incremental SCC detection [16, 21].

Incremental SCC detection has been studied on general graphs. (1) [22–25] study incremental SCC detection by maintaining the (weak) topological order of the nodes in the entire graphs. (2) [26–31] maintain the (weak) topological order of a directed acyclic graph (DAG) only, and terminate once an SCC is detected, which does not support continuously detecting and maintaining SCCs of general graphs. However, different from general graphs, edges in citation graphs typically follow an order of the published time of scholarly articles. Applying these methods directly to citation graphs leads to traversing 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 among the first to study incremental detection of SCCs tailored for citation graphs. The efficient detection on citation graphs essentially needs to deal with two issues. (1) How to employ the properties of citation graphs to incrementally detect SCCs, as all the existing SCC detection methods are for general graphs [22–32]. (2) How to design incremental algorithms bounded by the least amount of nodes as possible for handling continuously growing citation graphs [22, 25, 26].

Contributions. To this end, we propose efficient bounded incremental methods to detect strongly connected components on citation graphs in a dynamic environment.

(1) By exploiting the properties of different edge types, we first analyze how to partition the citation graph and maintain the local topological order to reduce the traversal of unnecessary nodes and edges for incremental methods (Section III).

(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 IV), based on the analyses in Section III.

(3) We design an efficient bounded batch incremental algorithm batDSCC by reducing the traversal of unnecessary edges (Section V), based on the 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 [33], DBLP [18], ACM [18] and MAG [17] (Section VI). We find that (a) our single incremental sinDSCC is on average $(12.6, 10.1, 50.7+, 76.4+, 19.5+)$ times faster than (AHRSZ [26], HKMST [25], incSCC$^+$ [23], MNR [29], PK$_2$ [31]), respectively. (b) Our batch incremental batDSCC is on average $(5.8, 5.0, 35.4+, 51.6+, 5.0)$ times faster than (AHRSZ, HKMST, incSCC$^+$, MNR, PK$_2$), respectively.

Detailed proofs and extra analyses are in the full version [34] due to the space limitations.

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 nodes 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.

(Weak) topological order [22, 25]. It is known that the SCCs can be ordered in a topological manner [35], and existing incremental algorithms are typically designed based on maintaining a (weak) topological order [22, 23, 25]. A *topological order* for a 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 [22]).

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}

Ordered list [36, 37]. 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 [36, 37]. (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.

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, $s \in K$ 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

Table II
STATISTICS OF REAL-LIFE CITATION GRAPHS (AAN [33], DBLP [18], ACM [18] AND MAG [17] ARE FOUR REAL-LIFE CITATION GRAPHS)

Datasets	$\|V\|$	$\|E\|$	$\frac{\|E_{n2o}\|}{\|E\|}$	$\frac{\|E_{s2s}\|}{\|E\|}$	$\frac{\|E_{o2n}\|}{\|E\|}$	|SCCS|	$\frac{\|\text{sizeof(SCC)=1}\|}{\|\text{SCCs}\|}$	$\frac{\|\text{sizeof(SCC)=2}\|}{\|\text{SCCs}\|}$	$\frac{\|\text{sizeof(SCC)>10}\|}{\|\text{SCCs}\|}$	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

$\|K_f\|$ = $|K_f| + |N^+(K_f)|$, 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 [30]. This is different from [26, 30] 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. ANALYSES OF INCREMENTAL DETECTION

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, 19] and the appends are common for incremental scholarly data analysis tasks [6, 7, 38], we focus on updates in an append manner in this study.

A. Properties of Citation Graphs

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

To do this, we divide the edges in E into three disjoint types, using $year$ as the default time granularity:
(1) $E_{n2o} = \{(u, v) | u.year > v.year \ and \ (u, v) \in E\}$, *i.e.,* a newly published article u cites an old published one v.
(2) $E_{s2s} = \{(u, v) | u.year = v.year \ and \ (u, v) \in E\}$, *i.e.,* an article u cites another one v published in the same year. We also denote E_{s2si} as the set of edges in E_{s2s} with year i only.
(3) $E_{o2n} = \{(u, v) | u.year < v.year \ and \ (u, v) \in E\}$, *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}. The time order of the nodes of this subgraph essentially forms 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 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}.* □

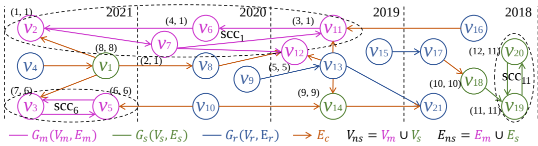
We perform a statistical analysis on four real-life citation graphs (AAN [33], DBLP [18], ACM [18] and MAG [17]), as illustrated in Table II. On citation graphs AAN, ACM and MAG, E_{n2o} occupies more than 95%, while E_{s2s} and E_{o2n} only account for less than 4% and 0.8%, respectively, and on DBLP, E_{n2o} 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. Partitioning of Citation Graphs

Different from existing incremental methods for general graphs [22, 23, 25, 32], they 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 are needed for citation graphs based on Properties 1 & 2. To do this, we partition the citation graph into disjoint subgraphs involved with different edge types E_{n2o}, E_{s2s} and E_{o2n}.

Partition. A citation graph $G(V, E)$ is *disjointedly* partitioned into three 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.
(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 $E_m = \{(x, y) | x \in V_m, y \in V_m, (x, y) \in E\}$. Here G_m contains all the SCCS having 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 $i \in [min\_year, max\_year]$. Not all edges of the 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.,* edge $(x, y) \in G_m$ when $x \in G_m$, $y \in G_r$. Indeed, there are only five types of edges $(x, y) \in E_c$: (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.



E_{n2o}: the set of edges ***left-to-right*** and crossing the vertical dashed lines, *e.g.*, (v_1, v_8).
E_{s2s}: the set of edges in the ***same-year***, *e.g.*, (v_1, v_3).
E_{o2n}: the set of edges ***right-to-left*** and crossing the vertical dashed lines, *e.g.*, (v_{10}, v_5).

Figure 2. Running example for the partition

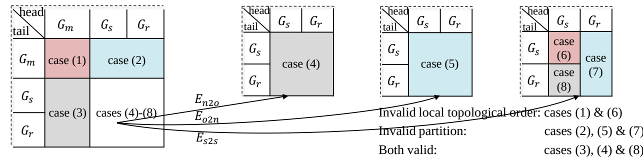


Figure 3. All cases of edge insertions

Graph $G(V, E)$ is partitioned such that $V = V_m \cup V_s \cup V_r$ and $E = E_m \cup E_s \cup E_r \cup E_c = E_{n2o} \cup E_{s2s} \cup E_{o2n}$. 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 2: We consider the citation graph G with 21 nodes with time attributes ($year$) and 26 edges, shown in Fig. 2.

After partitioning, G_m, G_s, G_r and E_c are colored purple, green, blue and orange, respectively. (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 of four subgraphs from year 2018 to 2021, where G_{s2018} has nodes $\{v_{18}, v_{19}, v_{20}\}$ and edges $\{(v_{18},\ v_{19}),\ (v_{19}, v_{20}), (v_{20}, v_{19})\}$. (3) G_r consists of the remaining nodes $\{v_4, v_8, v_9,\ v_{10},\ v_{13},\ v_{15}, v_{16},\ v_{17}, v_{21}\}$ and their connected edges. (4) E_c is a set of cross edges, including $(v_1, v_2), (v_1, v_3), (v_1, v_8),\ (v_4, v_1)$ and *etc*. □

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 subgraphs G_m and G_s only.* □

Proposition 1 can be proved by the definitions of subgraphs G_m and G_s, which 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.

Partition maintenance. We use two arrays inGm and inGsi to maintain the partition for continuously increasing nodes and edges, where (1) inGm is a flag map indicating whether a node is in G_m or not, and (2) inGsi maps a node to its associated $year$ if it is in G_{si}. 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. 3.

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 establish the Propositions for maintaining the partition for these cases.

Proposition 2: *The citation graph partition only needs to be adjusted for cases (2), (5), (7) when inserting edge (x, y).* □

Proposition 2 can be proved by the definition of the citation graph partition, which tells us that the 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 3: *For case (7), the partition is maintained by only marking y in G_s when inserting edge (x, y).* □

Proposition 3 is used to maintain the partition, resulting from a proof by contradiction that all edges of G_r are in E_{n2o}.

C. SCC *Detection for Citation Graphs*

After partitioning, SCC detection on citation graph G is reduced to find the SCCs of subgraphs G_m and G_s by Proposition 1. Below we first review the basic idea of a static method Pearce2 [39] to detect the SCCs of G_m and G_s, and how to maintain SCCs for handling incremental updates.

Static method Pearce2. Pearce2 is an up-to-date static SCC detection algorithm running in $\Theta(|V| + |E|)$ time, designed based on Tarjan [40]. It 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 id to each component of the graph.

Specifically, it maintains each node v a value $rindex$ to map each visited node to its component id, where the component root of SCC is found when $rindex$ has not changed after visiting all successors. As the DFS progresses, it recursively visits the successors w of node v, updates the $rindex$ of v with its smallest visit index of w, and pushes v into a stack. When $rindex$ remains unchanged, the component root is found, and all members of the SCC are popped out from the stack and assigned the same component id. Note that, the $rindex$ essentially forms the topological order of the graph as nodes of each SCC are popped from the stack recursively [39].

SCC **detection on subgraphs G_m and G_s**. (1) To detect the SCCs of G_m, one can call Pearce2 from all heads of E_{o2n}. (2) For each year $i \in [min\_year,\ max\_year]$, one can call Pearce2 from all heads of E_{s2si} by pruning the nodes visited by G_m to detect all SCCs of G_{si}. The SCCs of G_s are found by combining the SCC of each G_{si}. We refer this process to the method staDSCC, which runs 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. Besides, the arrays inGm and inGsi are assembled into staDSCC and initialized after it is finished.

We next illustrate algorithm staDSCC with an example.

Example 3: Consider Example 2 in Fig. 2 again. Assume that the nodes are visited in an ascending order, and each node is annotated with its visiting index and $rindex$. (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 staDSCC, where SCC_{19} with nodes $\{v_2, v_7, v_{11}, v_6\}$ is first detected. 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. Finally, SCC_{19}, SCC_{18} and SCC_{15} are the detected SCCs, surrounded by dashed lines. □

SCC **maintenance**. One can easily use an array mSCCs to store the set of SCCs, together with their respective members. To efficiently maintain SCC, a data structure called disjoint set dSet [35] is utilized to represent the nodes of SCCs for merging SCCs (using the union function of the disjoint set) and checking whether the nodes are in the same SCC (using the find function of the disjoint set). Both functions take $O(1)$ amortized time, and are fast.

The array mSCCs and disjoint set dSet are initialized with the SCCs of subgraphs G_m and G_s identified with staDSCC. That is, by using dSet we maintain a *dummy node* for each SCC and the *dummy node* represents the members of the SCC, such that the DAG representation of the citation graph is naturally obtained. In the sequel, the node with a prefix d stands for the dummy nodes of its represented SCC, by using find function of the disjoint set dSet.

D. Local Topological Order

We finally introduce local topological orders to reduce graph traversal costs. As explained before, incremental SCC detection 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 4: *A directed graph is a* DAG *if and only if it has a topological order [35].* □

Proposition 4 reveals that no extra SCCs are introduced when inserting edge (x, y) leads to a valid topological order, 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 5: *When inserting edges into a maintained partition, the local topological order needs to be adjusted for cases (1) & (6) only, which violates the local topological order.* □

Proposition 5 can be obtained by the definition of the local topological order and citation graph partition. This tells us that the local topological order of G remains valid and unaffected for cases (3), (4) & (8).

Local topological order maintenance. Inspired by [22, 25–28, 41], we use a set of ordered lists oLists to flexibly and efficiently reorder the local topological order, which is initialized by the $rindex$ of staDSCC as it essentially forms the topological order of G_m and each G_{si}, shown in Section III-C. Note that, oLists are maintained for the dummy nodes only, *i.e.,* the DAG representations of subgraphs G_m and each G_{si}. When creating a new order for a dummy node, the old order is first deleted using the delete function of ordered lists.

IV. 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 [22, 23, 25, 29, 31], and existing batch methods [31], [23] are based on single updates [29], [23], respectively. Based on our previous analyses, 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 1: *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., *$\|\mathsf{AFF}\| \le 2\|K_{min}\|$ 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 our incremental method needs to handle based on the analyses of incremental detection. Note that we handle insertions only in this study, as analyses show that scholarly data are rarely updated with deletions [2, 19].

(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, 2 and 5. 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 III are merged into 4 groups for ease of algorithm design by considering whether x and y in G_m, G_s, G_r or not, also illustrated in Fig. 3.

Group (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 2 and 5, which needs to be maintained based on [35].

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

Group (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, 2 and 5. Indeed, there is nothing to do for this group.

Group (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 group (c). For cases (5) & (7), the partition is invalid, which is the same as group (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 group (a).

For incremental updates, 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 maintained first, followed by the local topological order.

Data structures for single updates. Shared (used for all groups) and private (used for one group alone) data structures are included for single updates. Five shared data structures are utilized and initialized by staDSCC as analyzed in Section III: arrays inGm and inGsi to represent the partition, array mSCCs to represent the set of all SCCs, disjoint set dSet to represent the partition of the nodes defined by SCCs, and oLists to represent the local topological order. The private data structures will be introduced when handling each group.

B. Group (a): From G_m to G_m

We then present incremental method incGm2Gm to handle the group from G_m to G_m with invalid local topological orders, based on the analysis in Section III.

Algorithm incGm2Gm essentially performs on the DAG representation of G_m, shown in Fig. 4. 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. The detail of incGm2Gm is shown below.

Private data structures in incGm2Gm**.** (1) Array AFF stores the dummy nodes having invalid local topological orders, (2) arrays inF and inB indicate whether a dummy node of G_m is visited by forward or backward search, respectively.

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

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. $\quad numFE$ and $numBE$ are decreased by their smaller ones;
6. $\quad$ **if** $numFE = 0$ **then**
7. $\quad\quad$ AFF $\cup= \{f\}$; $forwPQ.pop()$;
8. $\quad\quad$ **for each** out edge (f, z) **do**
9. $\quad\quad\quad$ **if** $inB[dz]$ **then** /* SCC is detected */
10. $\quad\quad\quad\quad isExist = true$; $forwPQ.push(dz)$; $inF[dz] = true$;
11. $\quad\quad\quad$ **if** $!inF[dz]$ **then**
12. $\quad\quad\quad\quad forwPQ.push(dz)$; $inF[dz] = true$;
13. $\quad\quad$ **if** $forwPQ = \emptyset$ **then** $f = dx$; **else** $f = forwPQ.top()$;
14. $\quad\quad numFE = f$.outDegree;
15. $\quad$ **if** $numBE = 0$ **then**
16. $\quad\quad$ 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. $\quad$ Maintain mSCCs and dSet for SCCs;
4. **for each** dv **in** rTO **do**
5. $\quad$ oLists$[G_m]$.insertBefore$(ceiling[dv], dv)$.

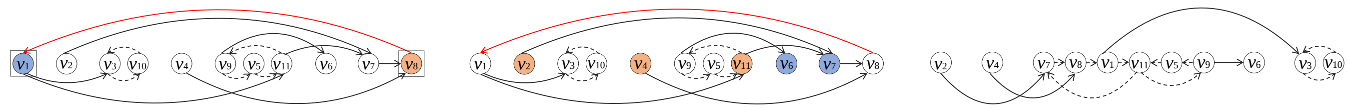
Figure 4. Algorithm incGm2Gm for group (a)

$numBE$, forward search is performed (lines 6-14). It updates AFF and $forwPQ$ as f violates the local topological order, then iteratively processes the out-neighbors z of f to check the existence of an SCC, and updates $forwPQ$ as the forward search candidates. (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 node of AFF using DFS based on [35], 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 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). (3) For each dummy node dv in rTO, its local topological order is maintained by the insertBefore function of ordered lists (lines 4, 5).

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 nodes



(a) Forward/backward search from dv_1/dv_8 (b) Procedure discover finished (c) Procedure maintain finished

Figure 5. Running example for incGm2Gm (All nodes and edges belong to G_m)

x and y are in the same SCC, or (dx, dy) has a valid local topological order, nothing needs to be done by Proposition 4 (line 2). (3) If (dx, dy) violates the 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 algorithm incGm2Gm with an example below.

Example 4: Consider inserting edge (v_8, v_1) into G_m with 11 nodes and 13 edges, and the inner edges of SCCs are indicated by dashed lines, shown in Fig. 5. The blue nodes denote the dummy nodes visited by forward search, and are in $forwPQ$. The orange nodes denote the dummy nodes visited by backward search, and are in $backPQ$. The forward or backward search is performed from the 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 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 discover and maintain. Procedure discover finds AFF $= \{dv_1, dv_8, dv_3, dv_{11}, dv_7\}$ and $isExist$ = true, as shown in Fig. 5(b). Procedure maintain detects the new SCC with dummy nodes $\{dv_1, dv_8, dv_{11}, dv_7\}$, maintains the local topological order and updates oLists, mSCCs, shown in Fig. 5(c). □

The correctness of incGm2Gm is assured as follows.

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

Proof Sketch: It is trivial for (x, y) in the same SCC or with a valid local topological order. We consider (x, y) violates the order, and show this from (1) AFF found 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) it correctly detects the SCCs if exists, and (4) it correctly maintains the partition and local topological order, see [34] for details. □

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, algorithm incGm2Gm is bounded by the minimum cover of affected node pairs K_{min} with $\|\mathsf{AFF}\| \leq 2\|K_{min}\|$, which takes the least amount of work when the out or in-neighbors of the cover are required to be traversed (see [34] for details).

C. Group (b): From G_m to $G_s \cup G_r$

We next present incremental algorithm incGm2Gsr to handle the group from G_m to $G_s \cup G_r$ with the invalid partition.

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 6. Algorithm incGm2Gsr for group (b)

Algorithm incGm2Gsr essentially performs on the DAG representation of G, shown in Fig. 6. It first maintains the partition using procedure scanGsGr1 to scan the subgraph $G_s \cup G_r$, and then detects the SCCs and maintains the local topological order with the help of algorithm incGm2Gm. The detail 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 $isVisit$.

(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

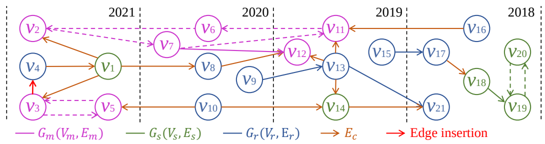


Figure 7. Running example for incGm2Gsr

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 [35]. (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 4. 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 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 illustrate algorithm incGm2Gsr with an example below.

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 3, where the edges of SCCs are drawn with dashed lines, shown in Fig. 7. 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 $ord(dv_3) \prec ord(dv_8) \prec ord(dv_2) \prec ord(dv_{12})$. $\square$

The correctness of incGm2Gsr is assured as follows.

Theorem 3: *Algorithm* incGm2Gsr *correctly detects the* SCCs *and maintains the partition and local topological order for group (b). Further,* AFF *is bounded,* i.e., $\|\text{AFF}\| \leq 2\|K_{min}\|$, *where K_{min} is the minimum cover of affected node pairs.* $\square$

Proof Sketch: We prove this by showing that it correctly (1) maintains the partition, (2) maintains the local topological order, and (3) detects the SCC if exists, provided in [34]. $\square$

Time complexity. Algorithm incGm2Gsr takes $O(\|\text{AFF}\| \, log\|\text{AFF}\| + |\text{AFFE}_\text{m}|)$ time, where AFF is a cover of the affected node pairs and bounded by K_{min} with $\|\text{AFF}\| \leq 2\|K_{min}\|$, and 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 $y \in G_s \cup G_r$, detailed analyses in [34].

D. Group (d): From $G_s \cup G_r$ to $G_s \cup G_r$

We then present incremental algorithm incGsr2Gsr to handle the group from $G_s \cup G_r$ to $G_s \cup G_r$ with invalid partition or local topological orders.

Algorithm incGm2Gm essentially performs on the DAG representation of G, shown in Fig. 8. For cases (4) & (8), the partition and local topological order are both valid and nothing needs to be done by Propositions 1, 2 & 5. 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 detail of incGsr2Gsr is shown below.

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 8. Algorithm incGsr2Gsr for group (d)

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, 2 & 5 (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 similarly to algorithm incGm2Gm, using $inSCC$ to find all members of the new SCC. Besides, it updates inGm to maintain the 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 3. Then, it creates dy an order lower than its out-neighbors in subgraph G_{si} to maintain the local

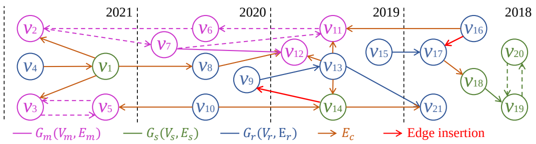


Figure 9. Running example for incGsr2Gsr

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 illustrate algorithm incGsr2Gsr with an example below.

Example 6: 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, as shown in Fig. 9. By calling scanGsGr2 to search all reachable dummy nodes from dv_9 on $G_s \cup G_r$, 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 $ord(dv_3) \prec ord(dv_9) \prec ord(dv_{21}) \prec ord(dv_{11}) \prec ord(dv_{12})$. □

The correctness of incGsr2Gsr is assured as follows.

Theorem 4: *Algorithm* incGsr2Gsr *correctly detects the* SCCs *and maintains the local topological order for group (d). Further,* AFF *is bounded,* i.e., $\|\mathsf{AFF}\| \leq 2\|K_{min}\|$, *where* K_{min} *is the minimum cover of affected node pairs.* □

Proof Sketch: 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 (see [34] for details). □

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 and bounded by K_{min} with $\|\mathsf{AFF}\| \leq 2\|K_{min}\|$, (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 $y \in G_s \cup G_r$, analyzed in [34].

E. The Complete Algorithm for Single Updates

We finally present the complete incremental SCC detection algorithm sinDSCC for single updates by combining the single node and edge insertions.

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 groups (a, b & d) from G_m to G_m, from 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 group (c) from $G_s \cup G_r$ to G_m by Propositions 1, 2 & 5.

Correctness. The correctness of sinDSCC follows easily from the results of staDSCC and the correctness of single node and edge insertions by Theorems 2, 3 & 4. Note that algorithm

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. Handle E_{invo} one by one with sinDSCC, similar to lines 6-10;
13. **return** updated G, mSCCs, inGm, inGsi, oLists, dSet.

Figure 10. Algorithm batDSCC for batch updates

sinDSCC is independent from the insertion order 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 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 $O(|V| + |E|)$.

Putting these together, we have proved Theorem 1.

Remarks. Different from (1) the incremental maintenance of (weak) topological order of a DAG [26–31] and (2) incremental SCC detection based on the (weak) topological order for the general graphs [22, 23, 25], our sinDSCC detects the SCCs, maintains the partition and local topological order for scholarly data. Our partition is also specifically designed based on the properties of citation graphs, which is also different from the partition for general graphs [42, 43].

V. 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 5: *Given the original partition and local topological order of citation graph $G(V, E)$ and an incremental 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.* □

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 detection efficiency.* □

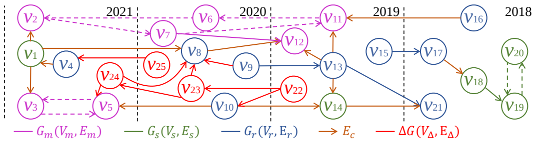


Figure 11. Running example for batch updates

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. 10, except that it handles 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 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, 12). (5) Finally, it returns the updated G, inGm, inGsi, mSCCs, oLists, and dSet (line 13).

We illustrate our batch update algorithm with an example.

Example 7: Consider inserting ΔG into G along the same drawing style as Example 5, as shown in Fig. 11. ΔG has 4 nodes, *i.e.,* $\{v_{22}, v_{23}, v_{24}, v_{25}\}$, and 8 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. The correctness of batDSCC follows from the one of sinDSCC and the fact that the results of sinDSCC are independent from the node and edge insertion orders.

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 space complexity of algorithm batDSCC is $O(|V| + |E| + |V_\Delta| + |E_\Delta|)$, dominated by the space of citation graphs.

Putting these together, we have proved Theorem 5.

VI. Experimental Study

Using four real-life citation graphs, we conduct experimental studies of our incremental methods sinDSCC and batDSCC.

A. Experimental Settings

Datasets. (1) AAN [33] 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. We compared our single and batch incremental algorithms sinDSCC and batDSCC with five competitive incremental methods: AHRSZ [26], HKMST [25], incSCC$^+$ [23, 24], MNR [29], $\mathsf{PK_2}$ [31]. (1) AHRSZ is a classic method to maintain the weak topological order of a DAG by iteratively searching from forward and backward. (2) HKMST detects the SCCs and maintains the topological order by using a two-way compatible search. (3) incSCC$^+$ handles batch updates for SCC detection by storing the shared affected areas that are obtained by forward and backward DFS. (4) MNR utilizes forward DFS to find the affected region to support incremental topological ordering. (5) $\mathsf{PK_2}$ presents a batch incremental topological ordering based on MNR to reduce redundant traversals. We extend AHRSZ, MNR and $\mathsf{PK_2}$ to support continuously detect and maintain SCCs, similar to batDSCC.

Implementation. All algorithms were implemented with C++, and graphs were represented by Boost Graph Library [44]. 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 single update algorithms. In the first 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 $\mathsf{PK_2}$. We randomly select $15,000$ edges as inserted edges.

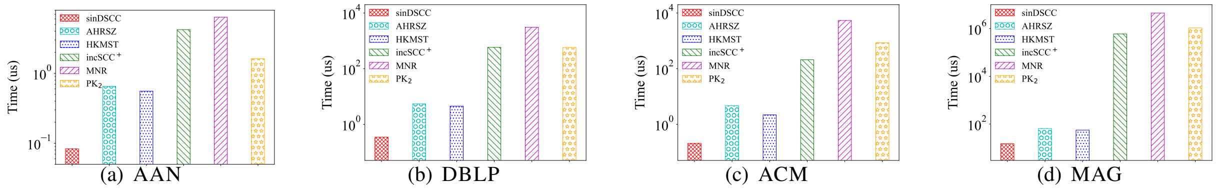


Figure 12. Efficiency tests of single update algorithms

Table III
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

Table IV
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	15395.0

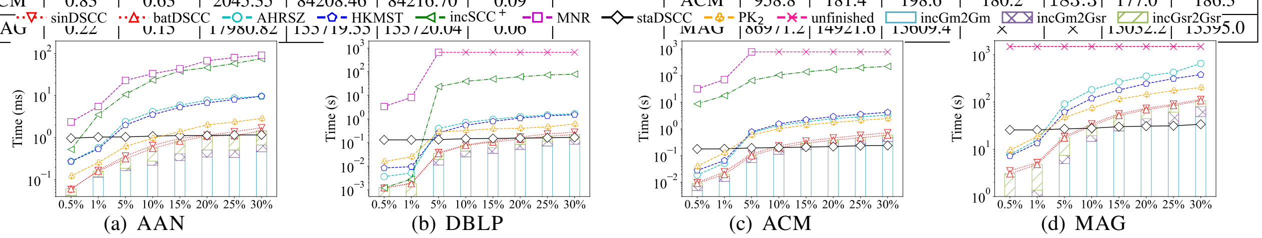


Figure 13. Efficiency tests of batch update algorithms

Table V
THE NUMBER OF EDGES PERFORMING GRAPH TRAVERSALS OF BATCH UPDATES (TO REFERS TO TOPOLOGICAL ORDER)

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

Exp-1.1. 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. 12.

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 incSCC$^+$, MNR and PK$_2$ are much larger than sinDSCC and each method requires to maintain the orders for all these affected nodes.

Exp-1.2. To evaluate the impact of single updates on the number of affected nodes, we summarize the average number of affected nodes per edge insertion for each incremental algorithm in Table III. All algorithms need 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, 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-2: Tests of batch update algorithms. In the second 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-2.1. 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. 13, 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 [22, 23, 25, 26]. 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 need 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. Note that this is particularly useful when the citation graphs update frequently.

Exp-2.2. 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 IV.

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 algorithm 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 algorithm batDSCC.

Exp-2.3. 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 V.

We find the following. First, sinDSCC and batDSCC both reduce at least (87%, 94%, 92%, 95%) invalid edges than their 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.

Summary. We have the following findings from these tests. (1) sinDSCC is on average $(12.6, 10.1, 50.7+, 76.4+, 19.5+)$ times faster than (AHRSZ, HKMST, incSCC$^+$, MNR, PK_2), respectively. (2) The intersection points of batDSCC and staDSCC are about $(25\%, 22\%, 10\%, 10\%)$ for (AAN, DBLP, ACM, MAG), respectively, and can fit for the need of the growing of scholarly data [3, 19]. Before the intersection points, batDSCC is $(5.8, 5.0, 35.4+, 51.6+, 5.0)$ times faster than (AHRSZ, HKMST, incSCC$^+$, MNR, PK_2), respectively.

VII. Related Work

Static SCC **detection**. Static SCC detection methods [39, 40, 45, 46] compute the SCCs for a given graph $G(V, E)$. Existing static methods all traverse the entire graph in a one or two pass manner, and take $\Theta(|V| + |E|)$ time.

Incremental SCC **detection**. Graphs are naturally dynamic and continuously growing, and incremental SCC detection methods [22–31] have been extensively studied.

Different from incremental methods in (1) [26–31] maintaining the (weak) topological order of a DAG only and (2) [22, 23, 25] for general graphs, 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 $\|\mathsf{AFF}\|$ [30] 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$^+$ [23, 24] 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.

Our incGm2Gm is inspired by AHRSZ [26]. 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 subgraphs, while AHRSZ performs on the entire graphs, resulting in unnecessary traversals. (3) Procedure maintain is simplified by removing the floor computation ($O(\|\mathsf{AFF}\| log \|\mathsf{AFF}\|)$ [26, 30]), and the number of newly created orders is relaxed to keep the total order of the nodes. Different from existing methods for general graphs, our study focuses on detecting SCCs in citation graphs, by exploiting the properties of scholarly data.

VIII. Conclusions

We have proposed bounded single and batch update algorithms sinDSCC and batDSCC to incrementally detect SCCs that support continuous updates of citation graphs. They are designed based on the analyses of citation graphs, graph partition and local topological order. We have also conducted extensive experiments to justify the advantages of our algorithms compared with their competitive algorithms.

A couple of topics need a further study. One is to handle deletions for scholarly data updates although this is a rare case [2, 19, 23, 47, 48], and the other is to clean incorrect citations based on detected SCCs to improve the quality of the scholarly data analytic tasks [49–51].

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

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

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

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

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

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

[29] 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.

[30] 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.

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

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

[33] 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.

[34] "Full-version," https://github.com/jfkey/detect-SCC/blob/main/full-version.pdf, 2023.

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

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

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

[38] 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.

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

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

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

[42] 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.

[43] W. Fan, M. Liu, C. Tian, R. Xu, and J. Zhou, "Incrementalization of graph partitioning algorithms," *Proc. VLDB Endow.*, vol. 13, no. 8, pp. 1261–1274, 2020.

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

[45] 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.

[46] 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.

[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.

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

[51] 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.

Supplementary Material

Appendix I: Real-life SCC *Example*

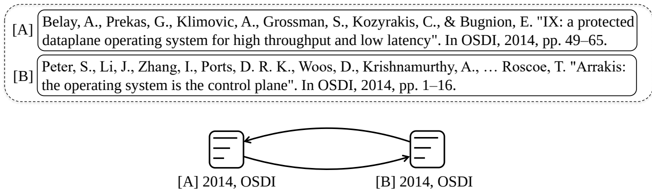


Figure 14. A real-life SCC formed by the time gap

The SCC formed by both the time gap and scholarly systems wrongly parsing has illustrated in Example 1 in Section I. Here we further show another real-life SCC example, formed due to the time gap between the accepted and published time of articles only.

Example 8: Fig. 14 demonstrates an SCC with two nodes formed by the *time gap* in the citation graph of DBLP [18], where nodes A and B represent two articles published at OSDI 2014. After the two articles are accepted, the committee finds the articles 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. □

Appendix II: Static SCC *Detection*

For a given citation graph, we next present the details of static method staDSCC, shown in Fig. 15, to detect SCCs by traversing from the edge heads of E_{o2n} and E_{s2s}, which is inspired by Pearce2 [39] and exploits the Properties 1 & 2.

The main result is stated below.

Theorem 6: *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 [39] is an up-to-date static strongly connected component detection algorithm running in $\Theta(|V| + |E|)$ time. It is designed based on Tarjan [40] 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 [39]. 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.

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.

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 15. Algorithm staDSCC

Procedure visit1 detects the SCCs involved 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 off 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 the 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.

Correctness. The correctness of algorithm staDSCC can be easily verified from the correctness of Pearce2 [39] 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 6 by the correctness and complexity analyses of algorithm staDSCC.

Remarks. Although Procedure visit1 of our static algorithm mostly comes from Pearce2 [39], 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.

Appendix III: Proofs on Partitioning

We present the detailed proofs of Propositions in Section III-B on partitioning citation graphs.

Proof of Proposition 1: 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 belong to G_m, and hence the SCC contains e must be defined in G_m. Second, for each edge $e \in E_{s2s}$ belonging to an SCC, there include two cases. (a) The SCC contains edges of E_{s2s}, E_{o2n} and E_{n2o}. (b) The SCC only contains edges of E_{s2s}. 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 on same-year subgraphs belong to G_{si}, and hence the SCC containing e must be defined in G_{si}. Hence, non-singleton SCCs exist in G_m and G_s only. □

Proof of Proposition 2: We show this from (1) algorithm staDSCC partitions the citation graph G into G_m, G_s, G_r and E_c, and (2) the definition of the citation graph partition.

(1) The partition and algorithm staDSCC are both designed based on the reachability with the same source nodes and pruning conditions. (a) 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. (b) For each year i, it traverses from the edge heads of E_{s2si} on the unvisited nodes of the same-year subgraphs via DFS, hence all nodes and edges of G_{si} are also found by staDSCC. (c) G_r and E_c can be easily obtained after G_m and each $G_{si} \in G_s$ are found. That is staDSCC partitions G into G_m, G_s, G_r and E_c.

(2) Based on the generated citation partition by staDSCC and its definition, 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. (a) 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. (b) For cases (5) & (7), the partition should also be adjusted by the definition of subgraphs G_m, G_s, respectively. □

Proof of Proposition 3: We show this from the definition of citation graph partition. First, G_m remains 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. □

Appendix IV: Proofs on Local Topological Order

We present the detailed proof of the Proposition in Section III-D on the local topological order.

Proof of Proposition 5: We show this from (1) algorithm staDSCC produces a local topological order of the citation graph G, and (2) the definition of the local topological order.

(1) As illustrated, we have generated the DAG representation of the graph after SCCs are detected. (a) 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. (b) 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.

(2) Based on the generated local topological order by staDSCC and its definition, the local topological order needs to be adjusted for cases (1) & (6), as the citation graph partition is already maintained. □

Appendix V: Correctness Proof of incGm2Gm

We present the detailed proof of Theorem 2 in Section IV-B on incremental algorithm incGm2Gm.

Proof of Theorem 2: 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) $\|\mathsf{AFF}\| \le 2\|K_{min}\|$, 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 $\|\mathsf{AFF}\| \le 2\|K_{min}\|$, 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, $\|\mathsf{AFF}\| \le 2\|K_{min}\|$ 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 2. (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 2. □

Appendix VI: Correctness Proof of incGm2Gsr

We present the detailed proof of Theorem 3 in Section IV-C on incremental algorithm incGm2Gsr.

Proof of Theorem 3: 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 2. 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 [35]. 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 3. □

Appendix VII: Correctness Proof of incGsr2Gsr

We present the detailed proof of Theorem 4 in Section IV-D on incremental algorithm incGsr2Gsr.

Proof of Theorem 4: 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, 2 & 5.

For case (5), (a) the partition of G_m, G_{si} and G_r can be easily maintained with the same analysis as Theorem 3. (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 3, 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 2 on G_{si}.

For case (7), (a) incGsr2Gsr marks y in G_{si}, then the partition is valid based on Proposition 3. The same analysis as Theorem 3, 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 4. □

Appendix VIII: Algorithm Complexity Analyses

We present the detailed time and space complexity analyses of our single sinDSCC and batch batDSCC incremental algorithms in Sections IV & V.

(1) Time complexity of single update algorithm incGm2Gm**.** The time cost arises from procedures discover and maintain. (a) 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. (b) 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 on the subgraph induced by AFF, $O(\|\mathsf{AFF}_b\|)$ 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, respectively, as each operation in the disjoint set takes $O(1)$ amortized time. Besides, at most $|\mathsf{AFF}|$ orders are created to maintain the local topological order, which takes $O(|\mathsf{AFF}|)$ time.

From these, our algorithm incGm2Gm takes $O(\|\mathsf{AFF}\| log \|\mathsf{AFF}\|)$ time for single edge insertions, where AFF is a cover of the affected node pairs of edge insertions. Note that, incGm2Gm is bounded by the minimum cover of affected node pairs K_{min} with $\|\mathsf{AFF}\| \leq 2\|K_{min}\|$, which takes the least amount of work when the out or in-neighbors of the cover are required to be traversed.

(2) Time complexity of single update algorithm incGm2Gsr**.** 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.

From these, 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, incGm2Gsr is bounded by the minimum cover of affected node pairs K_{min} with $\|\mathsf{AFF}\| \leq 2\|K_{min}\|$ and the affected edges $\mathsf{AFFE_m}$.

(3) Time complexity of single update algorithm incGsr2Gsr**.** The time cost can be attributed to (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 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.

From these, 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 $\mathsf{AFFE_m}$ of G_m, and the affected edges $\mathsf{AFFE_s}$ of G_s.

(4) Space complexity of single update algorithm sinDSCC**.** 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.

(5) Time complexity of batch update algorithm batDSCC**.** 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 $O(|V|+|V_\Delta|)$.

Appendix IX: Tests of Static Algorithm

We compare the running time and space cost of staDSCC with its competitors: Pearce2 [39], Tarjan [40], Gabow [45] and Kosaraju [46]. The impacts of $|E_{o2n}|$ and $|E_{s2s}|$ on the efficiency of static algorithms are also evaluated.

(1) Running time tests. 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 of efficiency tests are reported in Fig. 16.

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.

(2) Space cost tests. To evaluate the space cost, we test the memory cost on the entire datasets. The results are reported in Table VI.

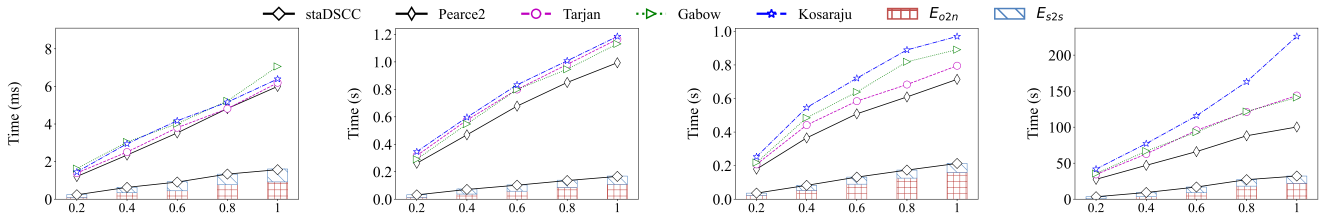
The space is mostly consumed by citation graphs, which accounts for (95%, 88%, 92%, 94%) on (AAN, DBLP, ACM, MAG) on average, respectively. The space cost of extra data structures of staDSCC is $(2.0,\ 4.2, 3.2, 5.4)$ times less than its competitors on (AAN, DBLP, ACM, MAG) on average, respectively. This is because staDSCC follows Pearce2 which reduces the space requirements, and staDSCC avoids unnecessary visits of nodes and edges.

(3) The efficiency of static algorithms *w.r.t.* the number of E_{o2n} and E_{s2s}. 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. 17.

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. 17(a) – 17(d).

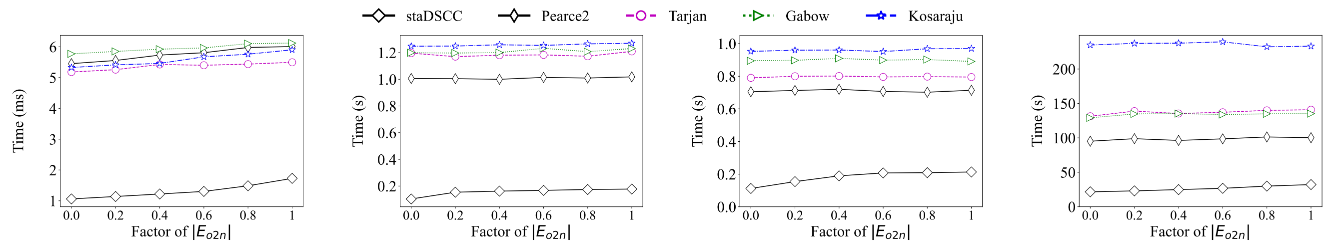
When varying $|E_{o2n}|$ with a factor from 0 to 1, our static staDSCC consistently runs faster than all static counterparts, and staDSCC is on average (4.2, 6.4, 4.1, 3.8) times faster than the best static competitors on (AAN, DBLP, ACM, MAG), respectively. Besides, when increasing the factor of $|E_{o2n}|$, 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% of 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

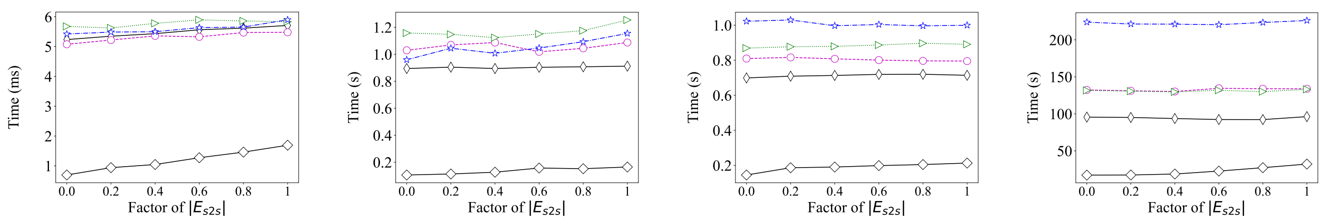


(a) AAN (b) DBLP (c) ACM (d) MAG

Figure 16. Efficiency tests of static SCC detection



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



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

Figure 17. Efficiency tests of static algorithms: varying E_{o2n} and E_{s2s}

Table VI
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

edges of E_{o2n} and E_{n2o} remain untouched in citation graphs. The results are reported in Figs. 17(e) – 17(h).

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.

Appendix X: Extra Tests of Incremental Algorithm

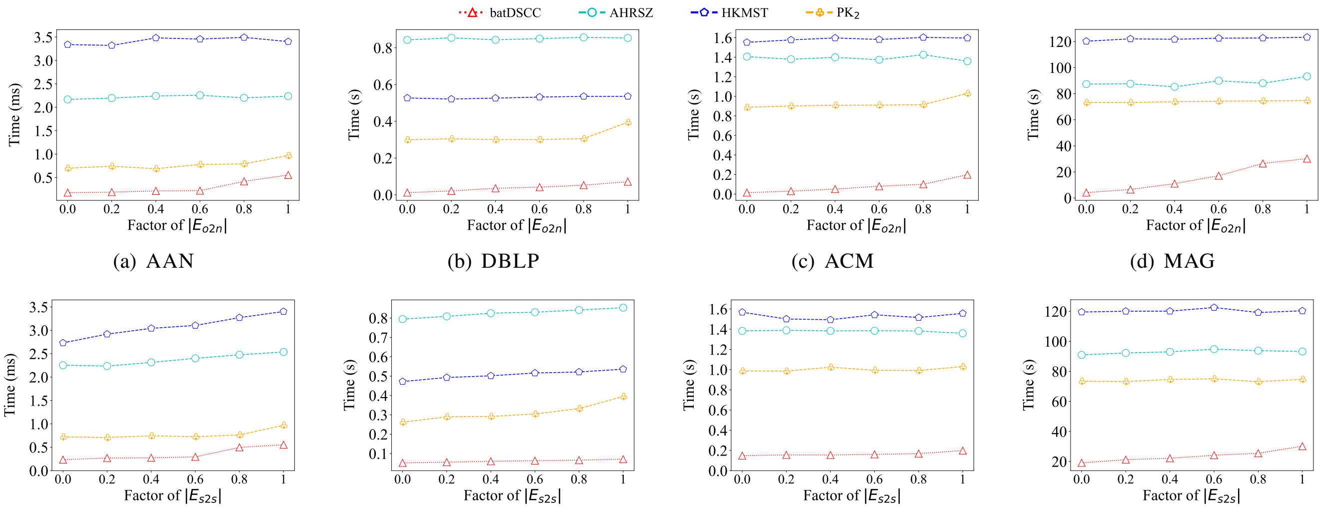
We evaluate the impacts of $|E_{o2n}|$ and $|E_{s2s}|$ on the efficiency of incremental algorithms to conduct the extra incremental algorithm tests of Section VI.

The efficiency of incremental algorithms *w.r.t.* the number of E_{o2n} and E_{s2s}. We separately study the impacts of $|E_{o2n}|$ and $|E_{s2s}|$ on the efficiency of incremental algorithms, *i.e.,* batDSCC, AHRSZ, HKMST, and PK_2. 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(a) – 18(d).

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,



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

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

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

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%

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(e) – 18(h).

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 $\|\mathsf{AFF}\|$.

Appendix XI: Impact of SCC on Scholarly Data Analysis Tasks

We study the impact of SCCs on computing PageRank scores for the article ranking task, as the detected SCCs can be utilized to clean incorrect citations to improve the quality of scholarly data.

The impact of SCCs **on PageRank Computation.** 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, 20]. 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.