Detecting Strongly Connected Components for Scholarly Data

Paper ID: 92

ABSTRACT

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

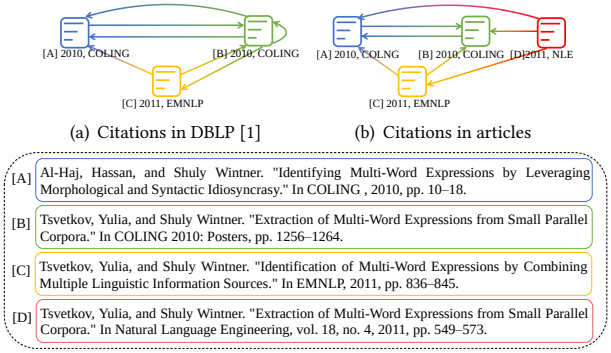
ACM Reference Format:

Paper ID: 92. 2023. Detecting Strongly Connected Components for Scholarly Data. In *Proceedings of International Conference on Management of Data (SIGMOD '23).* ACM, New York, NY, USA, 20 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Scholarly data has drawn significant attentions from both industry and academic communities for various research and technological analyses, among which citation graphs play an important role, such as scholarly article ranking [18, 26], scholarly community detection [16, 21, 28], scholar name disambiguation [25, 37], and scholarly search and analytic systems [24, 36, 38, 40].

Strongly connected components (SCCs) have significant impacts on the computational efficiency and data quality of citation graph based analyses. Specifically, scholarly article ranking commonly employs PageRank [6, 19, 26], where SCCs are detected first and each SCC is treated as a block in the process [6, 26]. SCCs of citation graphs are typically caused by two reasons. (1) There is a time gap between the accepted and published time of articles, and multiple articles cite each other. (2) Scholarly systems may wrongly treat the articles with similar or same titles as one article. SCC detection helps to identify SCCs caused by incorrect citations, and improve the quality of the scholarly data analysis tasks. A real-life SCC example of citation graphs is illustrated in Example 1.



(a) Citations in DBLP [1] (b) Citations in articles

Figure 1: A real-life SCC example

Example 1: Fig. 1(a) depicts an SCC in the citation graph of DBLP [1] formed due to the previous two reasons, where the nodes A, B and C are the articles published at COLING 2010, COLING 2010 and EMNLP 2011, respectively. (1) The edges (B, A) and (A, B) 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 to the formation of the SCC with nodes A, B and C. Note that article D is published at NLE 2011 that has the same title and authors as B, and D cites A, B and C, shown in Fig. 1(b). □

There exist quite a few studies designed for static and incremental SCC detection of general graphs. For instance, [10, 15, 29, 34, 39] study static SCC detection, and traverse the entire graphs using depth-first search (DFS); [5, 12, 17] study incremental SCC detection by incrementally maintaining the (weak) topological order of the nodes in the entire graphs. However, different from general graphs, edges in citation graphs typically follow an order of the published time of articles. Applying these methods directly to citation graphs leads to traverse unnecessary nodes and edges, which is too costly. That is, the properties of scholarly data should be exploited for the SCC detection of citation graphs.

To our knowledge, we are the first study on detecting SCCs specifically for citation graphs. However, the efficient SCC detection on citation graphs needs to deal with two issues. (1) How to employ the properties of citation graphs to detect SCCs, as all the existing SCC detection methods are for general graphs [2, 3, 5, 7, 8, 10, 12, 15, 17, 27, 29–31, 34, 39]. (2) How to incrementally detect SCCs, as citation graphs are typically large and continuously growing.

Contributions. To this end, we propose efficient static and incremental algorithms to detect SCCs in citation graphs.

(1) We develop an efficient static algorithm staDSCC to detect SCCs in citation graphs (Section 3), by dividing edges into three types and exploiting the properties of different types of edges to reduce the traversal of unnecessary nodes and edges.

SIGMOD '23, June 18–23, 2023, Seattle, WA, USA

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

(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 5), based on the analyses of incremental SCC detection (Section 4).

(3) We design an efficient bounded batch incremental algorithm batDSCC by reducing the traversal of unnecessary edges (Section 6), based on single update algorithm sinDSCC, to further improve the efficiency for continuous batch updates.

(4) Finally, we have conducted extensive experiments on four real-life citation graphs: AAN [32], DBLP [1], ACM [1] and MAG [36] (Section 7). We find that (a) our static algorithm staDSCC is both time and space efficient. Indeed, staDSCC is (4.9, 5.9, 6.0, 6.7) times faster and uses (2.0, 4.2, 3.2, 5.4) times less space than (Pearce [29], Tarjan [39], Gabow [15] and Kosaraju [34]) on average, respectively. (b) Our batch incremental algorithm batDSCC is on average (5.8, 5.0, 35.4+, 51.6+, 5.0) times faster than (AHRSZ [2], HKMST [17], incSCC$^+$ [12], MNR [27] and PK$_2$ [31]), respectively. (c) The SCCs have significant impacts on the PageRank scores, and the score gaps are on average (2.9%, 1.1%, 1.3%, 1.1%) on (AAN, DBLP, ACM, MAG) after removing SCCs, respectively.

Detailed proofs and extra analyses are in the Appendix due to space limitations.

2 PRELIMINARY

In this section, we introduce basic concepts.

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

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

(Weak) topological order [5, 17]. A *topological order* for a DAG is a *total order* "$<$" of its nodes. Generally, topological orders are not unique, and a topological order is valid if nodes $u < v$ 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 $u < v$ if (u, v) is an edge (that is, unreachable nodes may share the same order [5]).

Ordered list [4, 11]. 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 when inserting new orders. Ordered lists are proposed to solve this issue such that $ord(v)$ may be larger than $|V|$ for nodes 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), $ord(u) < ord(v)$, and maintains the orders. It supports four operations within $O(1)$ time [4, 11]. (1) insertAfter(u, v): insert item v immediately after item u in the topological order. (2) insertBefore(u, v): insert item v immediately before item u in the order. (3) order(u, v): determine whether item u precedes v in the order or not. (4) delete(u): delete item u from the order. Note that, when it performs operation insertAfter (or insertBefore)(u, v), only a new *order* is created after (or before) item u, and the topological orders of all nodes except v remain unchanged.

Table 1: 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 $V - V_m - V_s$ and their connected edges
E_c	Set of cross edges, *i.e.,* $E - E_m - E_s - E_r$
DAG	Directed acyclic graph

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 $s \rightsquigarrow t \wedge (ord(s) > ord(t) \vee s = t)$, where $s \rightsquigarrow t$ is a path on 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.

A cover K_{min} of the affected node pairs is minimum iff $\|K_{min}\| \leq \|K\|$ for all the other covers K. Here $\|K\|$ is the extended size of a cover K, which is the sum of the *extended-out size* of $\|K_f\|$ and the *extended-in size* of $\|K_b\|$ such that $\|K_f\| = |K_f| + |E^+(K_f)|$, where $K_f = \{s \in K | v \rightsquigarrow s\}$ and $E^+(K_f)$ is the set of out-neighbors of K_f, and $\|K_b\| = |K_b| + |E^-(K_b)|$, where $K_b = \{s \in K | s \rightsquigarrow u\}$ and $E^-(K_b)$ 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 [2, 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 of this study are summarized in Table 1.

3 STATIC SCC DETECTION

Citation graphs are typically large and continuously growing, hence incremental algorithms are essentially needed, which commonly employ the results of static algorithms. In this section, we first analyze the properties of citation graphs, and then present a static algorithm (an improved version of Pearce2 [29], by employing the properties of citation graphs) to facilitate the incremental detection of SCCs in citation graphs.

3.1 Analyses of Citation Graphs

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

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

Table 2: Statistics of real-life citation graphs

Datasets	|V|	|E|	$\|E_{n2o}\|/\|E\|$	$\|E_{s2s}\|/\|E\|$	$\|E_{o2n}\|/\|E\|$	|SCCs|	|SCCs=1|/|SCCs|	|SCCs=2|/|SCCs|	|SCCs>10|/|SCCs|	Largest SCC
AAN	18,041	82,897	95.92%	4.03%	0.05%	17,440	98.63%	1.13%	0.0170%	20
DBLP	3,140,081	6,149,832	77.00%	21.33%	1.68%	3,134,638	99.86%	0.12%	0.0004%	23
ACM	2,381,719	8,639,142	96.27%	2.95%	0.77%	2,373,013	99.72%	0.23%	0.0012%	40
MAG	183,685,867	730,799,115	97.71%	2.11%	0.19%	183,451,446	99.91%	0.16%	0.0004%	126

AAN [32], DBLP [1], ACM [1] and MAG [36] are four real-life citation graphs.

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

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

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

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

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

Assume that there exists a non-singleton subgraph is an SCC such that all its edges belong to E_{n2o}. 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 the SCCs. Further, there is no need to traverse the edges whose in-degrees of their tails are 0, as they cannot form any SCCs.

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

We perform a statistical analysis on four real-life citation graphs (AAN [32], DBLP [1], ACM [1] and MAG [36]), as illustrated in Table 2. On citation graphs AAN, ACM and MAG, E_{n2o} occupies more than 95%, while E_{s2s} and E_{o2n} only account for less than 4% and 0.8%, respectively, and on DBLP, E_{n2o} of DBLP occupies 77%, while E_{s2s} and E_{o2n} only account for 21.3% and 1.68%, respectively.

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

3.2 Algorithm for Static SCC Detection

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

The main result is stated below.

Theorem 3: *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.* □

We detect the SCCs involving within E_{o2n} and E_{s2s}, respectively, via DFS to generate a spanning forest, and each SCC is a subtree of the spanning forest. The root of the subtree is also known as the *component root* of the SCC. It identifies an SCC by finding the component root of the subtree in the spanning forest, based on Pearce2 [29]. We refer Pearce2 to Pearce for simplicity.

We next present the algorithm staDSCC shown in Fig. 2, where a detailed description is available in the Appendix.

Input: Citation graph $G(V, E)$, E_{o2n}, E_{s2s}, max_year, min_year.
Output: Each node's component identifier $rindex$, inG_m and inG_{si}.

1. Initialize $rindex[v] = 0$, $inG_m[v] = false$, $inG_{si}[v] = 0$ for all $v \in V$;
2. **for each** edge $e \in E_{o2n}$
3. **if** $rindex[e.head] = 0$ **then** visit1$(e.head, rindex, inG_m)$;
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$(e.head, i, rindex, inG_{si})$;
7. **return** $rindex$, inG_m, inG_{si}.

Figure 2: Algorithm staDSCC

Data structures. Our static algorithm staDSCC involves three main data structures, *i.e.,* arrays inG_m, inG_{si} and $rindex$. (1) inG_m is a flag map indicating whether a node is visited by the heads of edges in E_{o2n}. (2) inG_{si} maps a node to its associated $year$ if it is visited by the heads of edges in E_{s2s}. (3) $rindex$ maps each visited node to its component identifier.

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 node component identifier $rindex$, arrays inG_m and inG_{si}. (1) First, inG_m, inG_{si} and $rindex$ are initialized to $false$, 0 and 0 for each node v, respectively (lines 1). (2) It then calls procedure visit1 to detect all SCCs reachable from the heads of E_{o2n} for each unvisited head v (lines 2, 3). Specifically, visit1 traverses G using DFS to find the component roots of the nodes reachable from v, where $rindex$ and inG_m are set to its component identifier and $true$ for each visited node, respectively. (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). visit2 is a slight variant of visit1 that only traverses the unvisited nodes of the same-year subgraph in year i, where $rindex$ and inG_{si} are set to its component identifier and i for each visited node, respectively. (4) Finally, it returns $rindex$, inG_m, inG_{si} such that the set of SCCs of G can be assembled by iterating $rindex$ and gathering the nodes by the component identifier (line 7). Note that all SCCs of citation graph G are detected by only traversing from the edge heads of E_{o2n} and E_{s2s} by Property 1.

We next illustrate algorithm staDSCC with an example.

Example 2: We consider the citation graph G shown in Fig. 3, which contains 21 nodes and 26 edges, and each node has a time attribute, *i.e., year*. The nodes and edges colored purple and green indicate that they are visited from E_{o2n} and E_{s2s}, respectively. Assume that the nodes are visited in an ascending order, and each node is annotated with its visit order 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 then SCC_{19} with nodes $\{v_2, v_6, v_7, v_{11}\}$ is detected, and all its members are assigned with the same identifier 19. SCC_{18} with nodes $\{v_3, v_5\}$ is similarly detected by traversing from v_5. (2) For year 2021, $E_{s2s2021} = \{(v_1, v_2), (v_1, v_3), (v_3, v_5), (v_4, v_1), (v_5, v_3)\}$. Note that, v_2, v_3 and v_5 have been visited from E_{o2n}, and only node v_1 is visited from the head of edge (v_4, v_1). The E_{s2s}

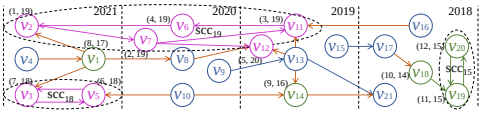


Figure 3: Running example for staDSCC

for years 2020, 2019 and 2018 is similarly traversed by staDSCC. Finally, SCC_{19}, SCC_{18} and SCC_{15} are the detected SCCs, surrounded by dashed lines. In fact, our staDSCC only visits 12 nodes and traverses 10 edges in the process, instead of the entire graph. □

Correctness. The correctness of algorithm staDSCC can be easily verified from the correctness of Pearce [29] and Property 1.

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. The space cost of staDSCC takes $O(|V|+|E|)$ space, among which the citation graph takes $O(|V|+|E|)$ space, and each of arrays $rindex$, inG_m and inG_{si} takes $O(|V|)$ space.

Finally, we have also proved Theorem 3 by the correctness and complexity analyses of algorithm staDSCC.

Remarks. In general, the incremental algorithms can be designed based on the results of any existing static algorithms [15, 29, 34, 39], once the topological orders of the nodes visited by staDSCC are obtained. Although our algorithm staDSCC is inspired by Pearce [29], 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 Pearce visits the entire graph in a one-pass manner.

4 ANALYSES OF INCREMENTAL DETECTION

In this section, to handle the large and continuously growing citation graphs, we analyze key design issues of incremental SCC detection solutions. As scholarly data are rarely updated with deletions, we focus on updates in an append manner.

4.1 Partitioning Citation Graphs

Existing incremental algorithms for SCC detection are for general graphs [3, 5, 12, 17], which traverse all the nodes/edges of the entire graph that are reachable from the nodes involved in the updates. However, not all nodes and edges of the citation graph of scholarly data are needed by Property 1, and Property 2 reveals that traversing edges in E_{o2n} and E_{s2s} further improves the efficiency (Section 3.1). To do this, we partition the citation graph based on the three disjoint types E_{n2o}, E_{s2s} and E_{o2n}.

Partition. A citation graph $G(V,E)$ is partitioned into three types of subgraphs G_m, G_s, G_r and a set E_c of cross edges, where we use *year* as the default time granularity.

(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 $E_m=\{(x,y)|x\in V_m, y\in V_m \textit{ and } (x,y)\in E\}$. Observe that G_m contains all the SCCs involved with the edges in E_{o2n}.

(2) Subgraph $G_s(V_s,E_s)$ consists of a set of subgraphs (one for each year). $G_s=\{G_{si}(V_{si},E_{si})|i\in[min\_year, max\_year]\}$, where min_year and max_year are the minimum and maximum year on citation graphs, respectively. For each $G_{si}(V_{si},E_{si})$, $V_{si}=V_{s2si}\cup V^+_{s2si}-V_m$, 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 $E_{si}=\{(x,y)|x,y\in V_{si} \textit{ and } (x,y)\in E\}$. 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 $V_r=V-V_m-V_s$, and $E_r=\{(x,y)|x, y\in V_r \textit{ and } (x,y)\in E\}$. Observe that G_r contains all the nodes of G except those in G_m and G_s.

(4) Edges $E_c=E-E_m-E_s-E_r$ is a set of cross edges among G_m, G_r, and G_{si} with $i\in[min\_year, max\_year]$. Indeed, there are 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.

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 of the tails are skipped when traversing from the edge heads of E_{o2n} and E_{s2s}.

We next illustrate the partition with an example.

Example 3: Consider Example 2 in Fig. 3 again. After partitioning, G_m, G_s, G_r and E_c are colored purple, green, blue and orange, respectively. (1) G_m has nodes $\{v_2,v_3,v_5,v_6,v_7,v_{11},v_{12}\}$ reachable from $\{v_2,v_5,v_6,v_{11}\}$. (2) $G_s=\{G_{s2021},G_{s2020},G_{s2019},G_{s2018}\}$, in which G_{s2018} has nodes $\{v_{18},v_{19},v_{20}\}$ and edges $\{(v_{18},v_{19}),(v_{19},v_{20}),(v_{20},v_{19})\}$. (3) G_r has nodes $\{v_4,v_8,v_9,v_{10},v_{13},v_{15},v_{16},v_{17},v_{21}\}$ and edges $\{(v_9,v_{13}),(v_{13},v_{21}),(v_{15},v_{17})\}$. And (4) E_c has edges $\{(v_1,v_2),(v_1,v_3),(v_1,v_8),(v_4,v_1),(v_8,v_{12}),(v_{10},v_5),(v_{10},v_{14}),(v_{13},v_{11}),(v_{13},v_{12}),(v_{13},v_{14}),(v_{14},v_{21}),(v_{16},v_{11}),(v_{17},v_{18})\}$. □

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

Proposition 4: *Non-singleton* SCCs *exist in* G_m *and* G_s *only.* □

Proposition 4 tells us that only G_m and G_s are needed for detecting all the non-singleton SCCs.

Proposition 5: *Algorithm* staDSCC *essentially partitions the citation graph* G *into* G_m, G_s, G_r *and* E_c. □

Proposition 5 tells us the result of staDSCC can be utilized to design incremental algorithms.

Partition maintenance. The partition needs to be maintained, as citation graphs are continuously updated over time.

Proposition 6: *The citation graph partition only needs to be adjusted when inserting edge* (x,y) *such that (1)* $x\in G_m$ *and* $y\in G_{si}\cup G_r$ *for any edge type, (2)* $x,y\in G_{si}\cup G_r$ *with* $(x,y)\in E_{o2n}$*, and (3)* $x\in G_{si}\cup G_r$ *and* $y\in G_r$ *with* $(x,y)\in E_{s2s}$ □

This tells us that the citation graph partition remains unaffected for the following six cases: (1) inserting a single node, (2) $x,y\in G_m$ for any edge type, (3) $x\in G_{si}\cup G_r$ and $y\in G_m$ for any edge type, (4) $x,y\in G_{si}\cup G_r$ with $(x,y)\in E_{n2o}$, (5) $x\in G_r$ and $y\in G_{si}$ with $(x,y)\in E_{s2s}$, and (6) $x,y\in G_{si}$ with $(x,y)\in E_{s2s}$.

Proposition 7: *When inserting edge* $(x,y)\in E_{s2s}$ *with* $x,y\in G_r$*, the partition is maintained by simply marking* y *in* G_{si}. □

Proposition 7 is used to maintain the partition, which is obtained by proof by contradiction that all edges of G_r are in E_{n2o}.

Remarks. (1) We can use flags for the nodes to easily maintain the partition. (2) Once the nodes of a subgraph are determined,

the edges can be easily determined, and the results inG_m and inG_s of staDSCC essentially denote the partition, where G_r contains the nodes not in G_m and G_{si}, and the remaining edges belong to E_c. (3) Different from the partition for general graphs [13, 14], our partition is based on the properties of citation graphs.

4.2 Maintaining SCCs

For incremental algorithms, efficient maintenance of SCCs is needed, and there are two main operations. First, SCCs may merge with each other to form bigger SCCs during the update process. Second, it needs to efficiently determine whether two nodes are in the same SCC. We next introduce how to efficiently maintain SCCs.

The data structure disjoint set [10] is utilized to represent the nodes of SCCs. Specifically, it maintains a *dummy node* for each SCC and the *dummy node* represents the members of the SCC, and we use dx to represent the dummy node of the SCC to which node x belongs. Two operations can be implemented by functions union and find of disjoint sets, respectively. (1) union(dx, dy): given two different *dummy nodes* dx and dy, it merges the two sets (*i.e.,* SCCs) containing nodes x and y, respectively, and makes dx as the dummy node of the new set (merged SCC). (2) find(x): given a node x, it returns its dummy node dx. Both functions take $O(1)$ amortized time [10]. Once SCCs are represented by dummy nodes, we naturally have a DAG representation of the citation graph.

Note that after static algorithm staDSCC detects the SCCs in the original citation graph, we have generated the DAG, represented by the disjoint set. For incremental algorithms, SCCs are essentially detected on DAGs, which are maintained during the process.

4.3 Local Topological Order

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

Proposition 8: *A directed graph is a* DAG *if and only if it has a topological order [10].* □

Proposition 8 reveals that no extra SCCs are introduced when inserting edge (x, y) has a valid topological order (*i.e.,* $ord(x) < ord(y)$), as the updated graph remains a DAG. That is, only the edges with invalid topological orders need to be handled.

Further, as shown by Proposition 4, we only need to deal with the topological orders of the DAG representations of subgraphs G_m and G_{si} ($i \in [min\_year, max\_year]$), 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 9: *Static algorithm* staDSCC *produces a local topological order of the citation graph G.* □

Proposition 9 reveals that staDSCC produces a local topological order ($rindex$), and can be utilized to design incremental algorithms.

Proposition 10: *Only when inserting edge (x, y) into a maintained partition with $x, y \in G_m$ or G_{si} that violates the local topological order, the local topological order needs to be adjusted.* □

Proposition 10 tells us that the local topological order of G remains valid and unaffected for cases (1), (3), (4) and (5) discussed in Proposition 6.

Local topological order maintenance. The local topological order is simply maintained for the dummy nodes only, *i.e.,* the DAG representations of subgraphs G_m and each $G_{si} \in G_s$. Indeed, all members of an SCC represented by a dummy node have the same order. Specifically, we use a set of ordered lists oLists to flexibly and efficiently reorder the local topological order, similar to [2, 5, 7, 8, 17, 20]. There are in total $(max\_year - min\_year + 2)$ ordered lists, where each ordered list maps a dummy node to a unique *order* to represent the topological order in the corresponding DAG.

5 DEALING WITH SINGLE UPDATES

In this section, we present our incremental algorithm to deal with single updates, as most existing incremental algorithms on general graphs are for single updates [5, 12, 17, 27, 31], and existing batch algorithms [31], [12] are based on single update algorithms [27], [12], respectively. Based on the previous analysis, our single update algorithm 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 11: *Given the original partition and local topological order of citation graph $G(V, E)$ and single node/edge insertions, there exists a bounded incremental algorithm that detects the* SCCs *and maintains the partition and local topological order of the updated G in $O(\|\mathsf{AFF}\| log \|\mathsf{AFF}\| + |\mathsf{AFFE_m}| + |\mathsf{AFFE_s}|)$ time, where (1)* AFF *is bounded,* i.e., $\|\mathsf{AFF}\| \leq 2\|K_{min}\|$ *such that K_{min} is the minimum cover of affected node pairs, and (2)* $\mathsf{AFFE_m}$ *and* $\mathsf{AFFE_s}$ *are the affected edges on G_m and on G_s, respectively.* □

5.1 Overview of Single Updates

We first discuss the types of single updates that incremental algorithms need to handle. Note that we handle insertions only in this study, as scholarly data are rarely updated with deletions.

(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 4, 6 and 10. Hence, x is simply marked in G_r.

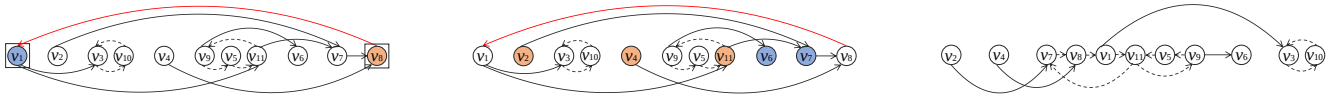
(2) Single edge insertions. When inserting an edge (x, y) with $x, y \in G_m$, G_{si} or G_r, there are 4 cases.

(a) From G_m to G_m. The partition is always valid, and the local topological order is invalid if $ord(x) > ord(y)$ by Propositions 6 and 10, which needs to be maintained based on [10].

(b) From G_m to $G_s \cup G_r$. The partition is invalid by Proposition 6, and the partition and local topological order need to be maintained.

(c) From $G_s \cup G_r$ to G_m. The partition and local topological order are both valid, and no extra SCCs are introduced by Propositions 4, 6 and 10. Indeed, there is nothing to do for this case.

(d) From $G_s \cup G_r$ to $G_s \cup G_r$. (i) When $(x, y) \in E_{n2o}$ or $(x, y) \in E_{s2s}$ with $x \in G_r$ and $y \in G_{si}$, the partition and local topological order are both valid, which is the same as case (c). (ii) When $(x, y) \in E_{o2n}$ or $(x, y) \in E_{s2s}$ with $x \in G_{si}$ and $y \in G_r$ or $(x, y) \in E_{s2s}$ with $x, y \in G_r$, the partition is invalid, which is the same as case (b). (iii) When $(x, y) \in E_{s2s}$ with $x, y \in G_{si}$, the partition is always valid,



(a) Insert (v_8, v_1) to G_m (b) Procedure discover finished (c) Procedure maintain finished

Figure 4: Running example for incGm2Gm

and the local topological order is invalid if $ord(x) > ord(y)$, which is the same as case (a).

After inserting edge (x, y), we say the edge is valid (resp. invalid) *w.r.t.* the partition if the partition is valid (resp. invalid), and similarly, we say the edge is valid (resp. invalid) *w.r.t.* the local topological order if the local topological order is valid (resp. invalid).

Note that the SCCs are affected and need to be detected when the partition or local topological order is invalid in the above cases.

Shared data structures for single updates. There are two classes of data structures: the outputs of static algorithm staDSCC and extra auxiliary data structures for incremental updates. (1) Arrays inG_m and inG_{si} maintain the partition of the citation graph, initialized by the outputs of staDSCC. When a node x belongs to a subgraph, *e.g.,* G_m or G_{si}, we also say that its dummy node dx belongs to the subgraph. (2) Auxiliary array memSCCs stores the members of each SCC, which is initialized by $rindex$ of staDSCC; Auxiliary disjoint set dSet represents the partition of the nodes defined by SCCs, which is used to efficiently find and merge SCCs; And $(max\_year - min\_year + 2)$ auxiliary ordered lists oLists are for efficiently maintaining the local topological order, where an ordered list maps each dummy node of an SCC to its unique order. Note that, the number of orders in oLists always equals to the number of dummy nodes in G_m and G_s, and for each newly created order, the old order is first deleted using the delete function of ordered lists.

For single updates, the partition is firstly maintained, followed by the local topological order. Observe that the nodes and edges of subgraph G_m increase only, while the nodes and edges of G_{si}, G_r and E_c may increase or decrease.

5.2 Case G_m to G_m

We then present incremental algorithm incGm2Gm that essentially performs on the DAG representation of G_m, shown in Fig. 5, to handle case G_m to G_m based on the analysis in Section 4.

(1) Dummy nodes dx, dy represent the SCCs to which x and y belong, respectively, (2) variable $isExist$ indicates whether a new SCC is introduced or not, (3) array AFF stores the dummy nodes having invalid local topological orders, (4) arrays inF and inB indicate whether a dummy node is visited by forward or backward search, respectively, for all dummy nodes in G_m, (5) rTO is the reverse local topological orders of the dummy nodes in the connected subgraph induced by AFF, and (6) for each dummy node dv in AFF, $Ceiling[dv]$ is the dummy node not in AFF with the lowest order when performing DFS on the nodes of AFF, where backtracing is triggered once it meets nodes not in AFF.

Procedure discover checks whether new SCCs exist and finds the dummy nodes with invalid local topological orders. Given input dummy nodes dx and dy, it updates AFF, inF, inB and $isExist$.

(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

Input: Edge (x, y) with $x, y \in G_m$, citation graph G, arrays inG_m, inG_{si}, memSCCs, ordered lists oLists, disjoint set dSet.
Output: Updated G, inG_m, inG_{si}, memSCCs, oLists, dSet.

1. Update G by inserting (x, y); dx = dSet.find(x); dy = dSet.find(y);
2. **if** $dx = dy$ **or** oLists$[G_m]$.order(dx, dy) **then return**
3. AFF $= \emptyset$; $inF[dv] = inB[dv] = false$ for all $v \in G_m$; $isExist = false$;
4. discover(dx, dy); maintain$(dx, dy, isExist,$ AFF$)$;
5. **return** updated G, inG_m, inG_{si}, memSCCs, 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. $u = min(numFE, numBE)$; $numFE -= u$; $numBE -= u$;
6. **if** $numFE = 0$ **then**
7. AFF $\cup= \{f\}$; $forwPQ.pop()$;
8. **for each** out edge (f, z) **do**
9. dz = dSet.find(z);
10. **if** $inB[dz]$ **then** /* SCC is detected */
11. $isExist = true$; $forwPQ.push(dz)$; $inF[dz] = true$;
12. **if** $!inF[dz]$ **then**
13. $forwPQ.push(dz)$; $inF[dz] = true$;
14. **if** $forwPQ = \emptyset$ **then** $f = dx$; **else** $f = forwPQ.top()$;
15. $numFE = f$.outDegree;
16. **if** $numBE = 0$ **then**
17. Do backward search similar to forward search (lines 7-15).

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

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

Figure 5: Algorithm incGm2Gm

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 4-17). $numFE$ and $numBE$ are decreased by their smaller ones (*i.e., u*), which determines to perform forward or backward search (line 5). (3) If the current $numFE$ is equal to or smaller than $numBE$, forward search is performed (lines 6-15). (a) The current f violates the local topological order, which is added into AFF, and removed from $forwPQ$ (line 7). (b) The out-neighbors z of f are then processed (lines 8-13), and dz is the dummy node to which SCC it belongs (line 9). If dz is already visited by backward search, $isExist$ is set to $true$ as there exists an SCC, dz is pushed into $forwPQ$, and $inF[dz]$ is set to $true$ (lines 10, 11). If dz is not visited by forward search, dz is simply pushed into $forwPQ$, and $inF[dz]$ is set to $true$ (lines 12, 13). (c) If $forwPQ$ is empty, f is set to dx, which terminates the while loop, and f is set to the top of $forwPQ$,

otherwise (line 14). (d) $numFE$ is updated to the out-degree of the current f (line 15). (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 16, 17).

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

(1) It first computes rTO and $Ceiling$ using DFS based on [10] (line 1). (2) If $isExist$ is true, memSCCs and dSet are maintained using DFS as well (lines 2, 3). The disjoint set dSet is updated by its operation union (dx, dy), and memSCCs is updated by merging the SCCs connected by dx and dy into a new SCC. Note that inserting edge (x, y) may lead to the merging of more than two SCCs. (3) For each dummy node dv in rTO, its local topological order is maintained by the insertBefore function of ordered lists (lines 4, 5). In fact, only the $Ceiling[dv]$ of the nodes dv in AFF reachable from dy need to be computed, as the $Ceiling[dv]$ for all nodes dv in AFF reachable to dx are $Ceiling[dy]$. Besides, the rTO of nodes dv for each node in AFF reachable from dy is obtained when computing $Ceiling[dv]$ using DFS, and the rTO for the nodes dv in AFF reachable to dx is computed using backward search.

Algorithm incGm2Gm takes as input an edge (x, y), citation graph G, arrays inG_m, inG_{si}, memSCCs, ordered lists oLists, disjoint set dSet, and returns the updated G, inG_m, inG_{si}, memSCCs, oLists, dSet. (1) It first updates G by inserting (x, y), and finds the dummy nodes dx and dy (line 1). (2) If $dx = dy$, nodes x and y are in the same SCC. If oLists$[G_m]$.order(dx, dy) holds, the local topological order for G_m remains valid, which means G_m remains a DAG by Proposition 8. Hence, nothing needs to be done in these cases (line 2). (3) If (dx, dy) violates the local topological order of G_m, it initializes AFF to empty, $isExist$ to false, inF and inB to $false$ for all dummy nodes of G_m, and calls procedures discover and maintain to detect the SCCs (lines 3, 4). (4) Finally, it returns the updated G, oLists, dSet, inG_m, inG_{si} and memSCCs (line 5).

We illustrate algorithm incGm2Gm with an example below.

Example 4: Consider inserting edge (v_8, v_1) into G_m that contains 11 nodes and 13 edges, shown in Fig. 4(a). incGm2Gm is actually performed on the dummy nodes of G_m, where the inner edges of SCCs are indicated by dashed lines. The blue nodes stand for the dummy nodes visited by forward search, and are in $forwPQ$. The orange nodes stand for the dummy nodes visited by backward search, and are in $backPQ$.

The forward or backward search is performed from the dummy nodes surrounded by a box, oLists of G_m saves the topological orders of its dummy nodes, *i.e.,* from left to right and in an increasing order, dSet maintains the dummy nodes of each SCC, *e.g.,* dv_1, dv_3, memSCCs stores the members of each SCC, *i.e.,* $\{\{v_3, v_{10}\}, \{v_5, v_9, v_{11}\}\}$, and inG_m and inG_{si} 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. 4(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, memSCCs, shown in Fig. 4(c). □

The correctness of algorithm incGm2Gm is assured as follows.

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

1. Update G by inserting (x, y); dx = dSet.find(x); dy = dSet.find(y);
2. $v_{lo} = +\infty$; $Rdy = \emptyset$; $isVisit[dv] = false$ for all $v \in G_s \cup G_r$;
3. scanGsGr1(dy, v_{lo});
4. Update inG_m and create a decreasing order for each node in Rdy;
5. **if** oLists$[G_m]$.order(dx, v_{lo}) **then return** ;
6. incGm2Gm$(dx, dy, G, inG_m, inG_{si}$, memSCCs, oLists, dSet$)$;
7. **return** updated G, inG_m, inG_{si}, memSCCs, oLists, dSet.

Procedure scanGsGr1(dy, v_{lo})

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

Figure 6: Algorithm incGm2Gsr

Theorem 12: *Algorithm* incGm2Gm *correctly detects the* SCCs *and maintains the partition and local topological order for case G_m to G_m. Further,* AFF *is bounded,* i.e., $\|AFF\| \leq 2\|K_{min}\|$, *where K_{min} is the minimum cover of affected node pairs.* □

Time complexity. Algorithm incGm2Gm takes $O(\|AFF\| log \|AFF\|)$ 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} as $\|AFF\| \leq 2\|K_{min}\|$, which takes the least amount of work when the out or in-neighbors of the cover are required to be traversed.

5.3 Case G_m to $G_s \cup G_r$

We next present incremental algorithm incGm2Gsr that essentially performs on the DAG representation of G, shown in Fig. 6, to handle case G_m to $G_s \cup G_r$ based on the analysis in Section 4.

(1) Dummy nodes dx, dy represent the SCCs to which x and y belong, respectively, (2) variable v_{lo} stores the current dummy node in G_m with the lowest order when performing DFS on $G_s \cup G_r$, where backtracing is triggered once nodes in G_m are met, (3) array Rdy stores the dummy nodes of $G_s \cup G_r$ reachable from dy with reverse local topological orders, and (4) array $isVisit$ indicates whether a dummy node of $G_s \cup G_r$ is visited by scanGsGr1 or not.

Procedure scanGsGr1 finds all reachable dummy nodes from dy on $G_s \cup G_r$. Given input 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 (lines 2-5). dz is the dummy node to which SCC it belongs (line 3), and 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 4). If $isVisit[dz] = false$ and $dz \in G_s \cup G_r$, it recursively calls scanGsGr1 with dz and v_{lo} (line 5). (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 6).

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

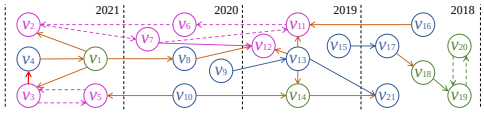


Figure 7: Running example for incGm2Gsr

scanGsGr1 with dy and v_{lo} and updates v_{lo}, Rdy and $isVisit$ (line 3). (3) For each dummy node dv in Rdy, it updates $inG_m[v] = true$ for each v with dummy node dv to maintain the partition. Besides, it creates a decreasing order using insertBefore that is lower than v_{lo} for each dv in Rdy (line 4). That is, the topological order of G_m without edge (dx, dy) is maintained [10]. (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 8. Hence, nothing needs to be done in this case (line 5). (5) If (dx, dy) violates the local topological order of G_m, it calls incGm2Gm with dx, dy, G, inG_m, inG_{si}, memSCCs oLists, dSet to detect the SCCs and maintain the local topological order (line 6). (6) Finally, it returns the updated G, inG_m, inG_{si}, memSCCs, oLists, and dSet (line 7).

We next illustrate incGm2Gsr with an example, shown in Fig. 7.

Example 5: Consider inserting edge (v_3, v_4) with $v_3 \in G_m$, $v_4 \in G_r$ into G along the same setting as Example 2, where the edges of SCCs are drawn with dashed lines. Besides, the topological orders of dummy nodes in G_m have $ord(dv_3) < ord(dv_2) < ord(dv_{12})$ 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 $ord(dv_4) < ord(dv_1) < ord(dv_8) < ord(dv_3) < ord(dv_2) < ord(dv_{12})$. 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 $ord(dv_3) < ord(dv_8) < ord(dv_2) < ord(dv_{12})$. □

The correctness of algorithm incGm2Gsr is assured as follows.

Theorem 13: *Algorithm* incGm2Gsr *correctly detects the* SCCs *and maintains the partition and local topological order for case G_m to $G_s \cup G_r$. Further,* AFF *is bounded,* i.e., $\|\text{AFF}\| \leq 2\|K_{min}\|$, *where K_{min} is the minimum cover of affected node pairs.* □

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 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} as $\|\text{AFF}\| \leq 2\|K_{min}\|$ and the affected edges AFFE_m.

5.4 Case $G_s \cup G_r$ to $G_s \cup G_r$

We next present incremental algorithm incGsr2Gsr that essentially performs on the DAG representation of G, shown in Fig. 8, to handle case $G_s \cup G_r$ to $G_s \cup G_r$ based on the analysis in Section 4.

(1) Dummy nodes dx, dy, variable v_{lo}, arrays Rdy, $isVisit$ are defined the same as incGm2Gsr, (2) variable $isExist$ indicates whether a new SCC is introduced or not, and (3) array $inSCC$ indicates whether a dummy node of $G_s \cup G_r$ is in the new SCC or not.

Procedure scanGsGr2 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

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

1. Update G by inserting (x, y); dx = dSet.find(x); dy = dSet.find(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 memSCCs and dSet for SCCs using $inSCC$;
9. Update inG_m and maintain the local topological order using Rdy;
10. **if** $(x, y) \in E_{s2s}$ **then**
11. **if** $x \in G_{si}$, $y \in G_{si}$ **then**
12. incGm2Gm$(dx, dy, G, inG_m, inG_{si}$, memSCCs, oLists, dSet$)$ on G_{si};
13. **if** $x \in G_r$, $y \in G_{si}$ **then return**
14. **if** $x \in G_r$, $y \in G_r$ **then**
15. Update inG_{si} and maintain the topological order for dy;
16. **if** $x \in G_{si}$, $y \in G_r$ **then**
17. Repeat lines 15 and 12;
18. **return** updated G, inG_m, inG_{si}, memSCCs, oLists, dSet.

Figure 8: Algorithm incGsr2Gsr

line 5 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 algorithm 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 $(x, y) \in E_{n2o}$, then the partition and local topological order remain valid, which means that (x, y) does not introduce any SCCs by Propositions 4, 6 & 10, and nothing needs to be done in this case (line 4). (3) If $(x, y) \in E_{o2n}$, it calls scanGsGr2 with dy, v_{lo}, and updates v_{lo}, Rdy, $isVisit$, $isExist$, and $inSCC$. If $isExist = true$, memSCCs and dSet are maintained similar to algorithm incGm2Gm, using $inSCC$ to find all members of the new SCC. Besides, it updates inG_m 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 four cases for x and y in G_{si} or G_r (lines 10-18). (a) If $x \in G_{si}$ and $y \in G_{si}$, it calls incGm2Gm with dx, dy, G, inG_m, inG_{si}, memSCCs 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 $x \in G_r$ and $y \in G_{si}$, it does nothing, similar to the case of $(x, y) \in E_{n2o}$ (line 13). (c) If $x \in G_r$ and $y \in G_r$, it only updates $inG_{si}[y]$ to $y.year$ to maintain the partition. Then, it creates dy an order lower than its out-neighbors in subgraph G_{si} to maintain the local topological order. Note that, SCCs are not introduced in this case by Proposition 7 (lines 14, 15). (d) If $x \in G_{si}$ and $y \in G_r$, it repeats line 15 to maintain the partition and the local topological order, except (dx, dy). It then repeats line 12 to detect the SCCs and maintain the local topological order (lines 16, 17). (5) Finally, it returns the updated G, inG_m, inG_{si}, memSCCs, oLists, and dSet (line 18).

We next illustrate incGsr2Gsr with two examples in Fig. 9.

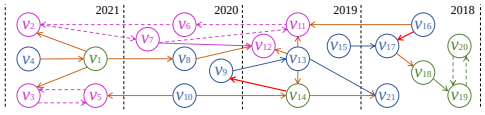


Figure 9: Running example for incGsr2Gsr

Example 6: (1) Consider inserting edge $(v_{14}, v_9) \in E_{o2n}$ with $v_{14} \in G_{s2019}$ and $v_9 \in G_r$ along the same drawing style as Example 5. Algorithm incGsr2Gsr detects the SCC with $\{dv_9, dv_{13}, dv_{14}\}$, and updates v_{lo} to dv_{11} by calling scanGsGr2. 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 $ord(dv_3) < ord(dv_9) < ord(dv_{21}) < ord(dv_{11}) < ord(dv_{12})$.

(2) Consider inserting edge $(v_{16}, v_{17}) \in E_{s2s}$ with $v_{16}, v_{17} \in G_r$ in the same drawing style as Example 5. incGsr2Gsr updates v_{17} in G_{s2018} and creates an order lower than dv_{18}. There are no introduced SCCs, and the partition and local topological order of G_{s2018} are valid with $ord(dv_{17}) < ord(dv_{18}) < ord(dv_{19})$. □

The correctness of algorithm incGsr2Gsr is assured as follows.

Theorem 14: *Algorithm* incGsr2Gsr *correctly detects the* SCCs *and maintains the local topological order for case $G_s \cup G_r$ to $G_s \cup G_r$. Further,* AFF *is bounded,* i.e., $\|\text{AFF}\| \leq 2\|K_{min}\|$, *where K_{min} is the minimum cover of affected node pairs.* □

Time complexity. Algorithm incGsr2Gsr takes $O(\|\text{AFF}\|log\|\text{AFF}\| + |\text{AFFE}_\text{m}| + |\text{AFFE}_\text{s}|)$ time, where (1) AFF is a cover of the affected node pairs, (2) AFFE_m is the affected edges (*i.e.,* those added edges) on G_m, and (3) 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 $\|\text{AFF}\| \leq 2\|K_{min}\|$, the affected edges AFFE_m in G_m, and the affected edges AFFE_s in G_s,

5.5 The Complete Algorithm

We finally present the complete incremental SCC detection algorithm sinDSCC for single updates by combining the single node and edge insertions (algorithms incGm2Gm, incGm2Gsr and incGsr2Gsr).

Algorithm sinDSCC takes as input a node x or an edge (x, y), citation graph G, arrays inG_m, inG_{si}, memSCCs, ordered lists oLists and disjoint set dSet, and returns the updated G, inG_m, inG_{si}, memSCCs, oLists, dSet. (1) For a single node insertion x, it simply marks the node x in G_r. (2) For a single edge insertion (x, y), it calls algorithms incGm2Gm, incGm2Gsr and incGsr2Gsr for cases G_m to G_m, G_m to $G_s \cup G_r$ and $G_s \cup G_r$ to $G_s \cup G_r$, respectively, and does nothing for case $G_s \cup G_r$ to G_m by Propositions 4, 6 and 10.

Correctness. The correctness of algorithm sinDSCC follows easily from the results of staDSCC based on Propositions 5 & 9 and the correctness of single node and edge insertions by Theorems 12, 13 & 14. Note that the correctness of sinDSCC is independent of the insertion sequence of nodes and edges, as before and after single insertions, the partition and local topological order remain valid.

Time and space complexities. By the time complexity analyses of algorithms incGm2Gm, incGm2Gsr and incGsr2Gsr, it is easy to know that sinDSCC is bounded in $O(\|\text{AFF}\|log\|\text{AFF}\| + |\text{AFFE}_\text{m}| + |\text{AFFE}_\text{s}|)$ time for single updates. The space complexity of algorithm sinDSCC is $O(|V| + |E|)$, dominated by its key data structures.

Putting these together, we have proved Theorem 11.

Input: Batch updates $\Delta G(V_\Delta, E_\Delta)$, citation graph G, arrays inG_m, inG_{si}, memSCCs, ordered lists oLists, disjoint set dSet.
Output: Updated G, inG_m, inG_{si}, memSCCs, 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** $E_{invp} = E_\Delta - E_{vp}$;
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 $E_{vp} \cap E_{vo}$; $E_{invp} = E_\Delta - E_{vp}$;
10. **let** $E_{invo} = E_\Delta - E_{vo}$;
11. **while** $E_{invo} \neq \emptyset$ **do**
12. Handle the first edge in E_{invo} with sinDSCC;
13. Append newly valid edges to E_{vo};
14. Update G by inserting valid edges E_{vo}; $E_{invo} = E_\Delta - E_{vo}$;
15. **return** updated G, inG_m, inG_{si}, memSCCs, oLists, dSet.

Figure 10: Algorithm batDSCC

Remarks. Different from (1) the incremental maintenance of (weak) topological order of a DAG [2, 7, 8, 27, 30, 31] and (2) incremental SCC detection based on the (weak) topological order for the general graphs [5, 12, 17], our sinDSCC detects the SCCs, maintains the partition and local topological order for scholarly data.

6 DEALING WITH BATCH UPDATES

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

The main result is stated below.

Theorem 15: *Given the original partition and local topological order of citation graph $G(V, E)$ and an increment subgraph $\Delta G(V_\Delta, E_\Delta)$, there exists a bounded incremental algorithm that detects the* SCCs *and maintains the partition and local topological order of $G + \Delta G$ in $O(|\text{AFFE}_\text{m}| + |\text{AFFE}_\text{s}| + |V_\Delta| + |E_\Delta|\|\text{AFF}\|log\|\text{AFF}\|)$ time, where (1)* AFF *is bounded,* i.e., $\|\text{AFF}\| \leq 2\|K_{min}\|$ *such that K_{min} is the minimum cover of affected node pairs, and (2)* AFFE_m *and* 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. Indeed, algorithm sinDSCC simply inserts the valid edges *w.r.t.* the partition and local topological order without any graph traversal, and we have the following observation.

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

Observe that (1) valid edges *w.r.t.* the partition and local topological order may become invalid when sinDSCC handles invalid edges. (2) When sinDSCC maintains the partition, the invalid edges *w.r.t.* the local topological order may change. These tell us that valid edges should be considered first, and invalid edges *w.r.t.* the partition should be considered before those *w.r.t.* the local topological order. In this way, invalid edges are reduced.

Algorithm batDSCC takes the same input and output as algorithm sinDSCC shown in Fig. 10, except that it has batch updates

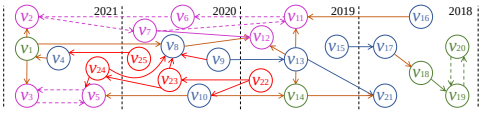


Figure 11: Running example for batch updates

$\Delta G(V_\Delta, E_\Delta)$. (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.,* $E_\Delta - E_{vp}$ is processed one by one until E_{invp} is empty (lines 5-9). Note that the invalid edges of E_{invp} in G_m are processed first. (a) It handles the first edge in E_{invo} 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 $E_\Delta - E_{vp}$ (line 9). (4) The invalid edges *w.r.t.* the local topological orders are processed similarly (lines 10-14). (5) Finally, it returns the updated G, inG_m, inG_{si}, memSCCs, oLists, and dSet (line 15).

We next illustrate batch updates with an example in Fig. 11.

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

Correctness. Algorithm batDSCC is essentially designed based on sinDSCC and utilizes the results of staDSCC by Propositions 5 & 9. Besides, for any sequences of batch updates, the SCCs can be correctly detected by handling updates one by one with sinDSCC. Actually, batDSCC first updates nodes and valid edges $E_{vp} \cap E_{vo}$. It then handles E_{invp} one by one with sinDSCC and updates the newly valid edges $E_{vp} \cap E_{vo}$ until E_{invp} is empty. It also handles E_{invo} one by one with sinDSCC and only updates the newly valid edges E_{vo} until E_{invo} is empty, where the partition is always valid for E_{invo}. That means algorithm batDSCC handles all updates in ΔG one by one with sinDSCC, which assures the correctness.

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

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

Putting these together, we have proved Theorem 15.

7 EXPERIMENTAL STUDY

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

7.1 Experimental Settings

Datasets. (1) AAN [32] collects the articles published at ACL from 1965 to 2011. (2) DBLP [1] collects the publications at the DBLP Bibliography from 1936 to 2016. (3) ACM [1] also collects in Computer Science from 1936 to 2016. (4) MAG [36] 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 2.

Algorithms. (1) We compared our static staDSCC with four competitive methods: Pearce [29], Tarjan [39], Gabow [15] and Kosaraju [10, 34]. (2) We compared our single and batch incremental algorithms sinDSCC and batDSCC with five competitive incremental methods: AHRSZ [2], HKMST [17], incSCC$^+$ [12], MNR [27], $\mathsf{PK_2}$ [31]. As algorithms AHRSZ, MNR and $\mathsf{PK_2}$ only incrementally maintain the (weak) topological order of a DAG, and terminate once an SCC is detected, we extended them to support continuously detect and maintain SCCs, similar to our batDSCC and staDSCC.

Implementation. All algorithms were implemented with C++, and graphs were represented by Boost Graph Library [35]. 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://anonymous.4open.science/r/detect-SCC-3FEB.

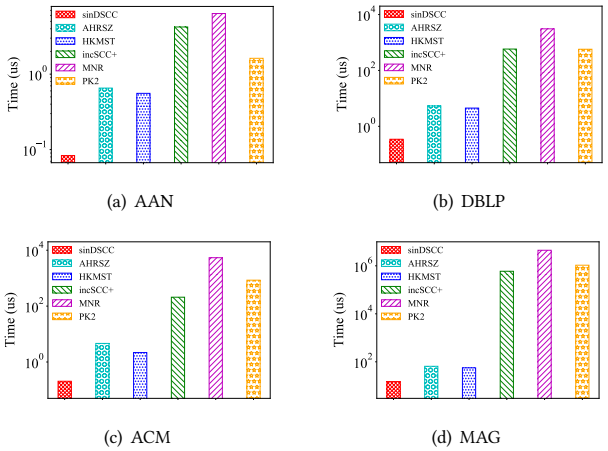
7.2 Experimental Results

We next present our findings.

Exp-1: Tests of static algorithms. In the first set of tests, we compare the running time and space cost of staDSCC with its competitors. Specifically, staDSCC is (4.9, 5.9, 6.0, 6.7) times faster and uses (2.0, 4.2, 3.2, 5.4) times less space than (Pearce, Tarjan, Gabow and Kosaraju) on average, respectively. Detailed and extra tests of static algorithms are in the Appendix due to space limitations.

Exp-2: Tests of single update algorithms. In the second set of tests, we compare the running time and space cost of single update algorithm batDSCC with AHRSZ, HKMST, incSCC$^+$, MNR and $\mathsf{PK_2}$. We randomly select 15, 000 edges as inserted edges. The average time per edge insertion is reported in Fig. 12.

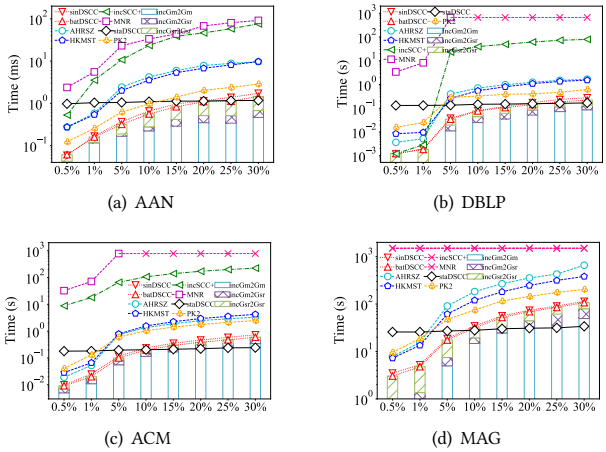
sinDSCC consistently runs faster than its single update competitors, and is on average (7.9, 15.5, 22.6, 4.3), (6.7, 13.0, 16.9, 3.7), (50.7, 1682.6, 1018.2, 39437.9), (76.4, 8903.0, 26371.1, 295953.2), (19.5, 1658.2, 4147.6, 70562.0) times faster than AHRSZ, HKMST, incSCC$^+$, MNR, $\mathsf{PK_2}$ on (AAN, DBLP, ACM, MAG), respectively. The rationale behind this lies in that the affected nodes found by algorithms incSCC$^+$, MNR and $\mathsf{PK_2}$ are much larger than sinDSCC, as shown



(a) AAN (b) DBLP

(c) ACM (d) MAG

Figure 12: Efficiency tests of single update algorithms



(a) AAN (b) DBLP

(c) ACM (d) MAG

Figure 13: Efficiency tests of batch update algorithms

Table 3: Space cost of incremental algorithms (MB)

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

in Table 7 in the Appendix, and each algorithm requires to maintain the orders for all these affected nodes.

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

Exp-3.1. To evaluate the impacts of $|\Delta G|$, we vary $|\Delta G|$ from 0.5% to 30% of $|G|$, *i.e.,* from 319 to 19.13K, 33.22K to 1.99M, 23.65K to 1.42M, and 2.81M to 168.65M on AAN, DBLP, ACM and MAG, respectively. The results are reported in Fig. 13, where those incremental algorithms 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. 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. batDSCC is on average (4.5, 8.1, 30.1, 5.2) times faster than staDSCC on (AAN, DBLP, ACM, MAG), respectively.

Note that batDSCC and all known incremental algorithms are not consistently faster than static algorithms [2, 5, 12, 17]. Based on its time complexity $O(|\mathsf{AFFE_m}|+|\mathsf{AFFE_s}|+|V_\Delta|+|E_\Delta|\|\mathsf{AFF}\|log\|\mathsf{AFF}\|)$, where $|\mathsf{AFFE_m}|+|\mathsf{AFFE_s}|+|V_\Delta|$ is consistently less than $|V_{ns}|+|E_{ns}|$. The choice between batDSCC and staDSCC mainly depends on $\|\mathsf{AFF}\|$ and $|E_\Delta|$ whether satisfies $c|E_\Delta|\|\mathsf{AFF}\|log\|\mathsf{AFF}\|$ is less than $|V_{ns}|+|E_{ns}|$, where c is a const factor related to the complexity.

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

When varying $|\Delta G|$ from 0.5% to 30%, algorithms incGm2Gm, incGm2Gsr and incGsr2Gsr in batDSCC all increase with the increment of $|\Delta G|$. Algorithms incGsr2Gsr and incGm2Gm take the most and second most time, which occupy (45%, 53%, 56%, 48%) and (45%, 27%, 17%, 24%) on (AAN, DBLP, ACM, MAG), 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. This is consistent with the analyses of batch updates in Section 6.

Exp-3.2. To evaluate the space cost of incremental algorithms, we test the memory cost in practice when fixing $|\Delta G|$ = 30%. The space cost analyses include the consumption of citation graphs and data structures, and the results are reported in Table 3.

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

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

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

Table 4: 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%

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 [26] to compute the PageRank scores of citation graphs, where each SCC is treated as a block in the process [6, 26]. For the PageRank algorithm of [26], 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 4.

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

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

8 RELATED WORK

Static SCC **detection**. Static SCC detection methods [15, 29, 34, 39] compute the SCCs for a given graph $G(V, E)$, which are also known as batch methods. Existing static methods all traverse the entire graph in a one or two pass manner, and take $O(|V| + |E|)$ time. Tarjan [39] employs the property that nodes of an SCC form a subtree in the spanning tree of the graph, and detects SCCs via repeated DFS over the entire graph in a one-pass manner. Based on the idea of Tarjan, Gabow [15] and Pearce [29] present one-pass algorithms to reduce the space requirements. Moreover, Kosaraju [34] employs the property that the transpose and original graph share the same SCCs, and it detects SCCs in two-pass via DFS on the two graphs. Differently, our static algorithm divides edges into three types and exploits the properties of different types of edges to reduce the traversal of unnecessary nodes and edges.

Incremental SCC **detection**. Graphs are naturally dynamic and continuously growing, and incremental SCC detection methods [2, 5, 7, 8, 12, 17, 27, 30, 31] have been extensively studied.

(1) Algorithms in [2, 7, 8, 27, 30, 31] maintain the (weak) topological order of a DAG only, and terminate once an SCC is detected. AHRSZ [2], MNR [27], PK [30] and PK$_2$ [31] can be extended to support continuously detect and maintain SCCs, similar to our staDSCC and batDSCC. However, BC [7] and BK [8] use a different concept of node equivalence to maintain the topological order, which can not be extended. (2) Algorithms in [5, 12, 17] detect and maintain SCCs for general graphs. BFGT [5] and HKMST [17] take $O(|E|^{3/2})$ time for inserting $|E|$ edges one by one into graphs with $|V|$ nodes. incSCC$^+$ [12] presents a bounded relative to a batch algorithm, and defines an affected region based on Tarjan [39]. Further, BFGT [5] depends on the level status of each node generated when inserting edges, and can hardly be used to incrementally detect SCCs as they do not support the batch update of the level status.

Different from incremental methods in (1) [2, 7, 8, 27, 30, 31] and (2) [5, 12, 17] for general graph, our batDSCC takes $O(|\mathsf{AFFE_m}| + |\mathsf{AFFE_s}|+|V_\Delta|+|E_\Delta|\|\mathsf{AFF}\|log\|\mathsf{AFF}\|)$ time for batch updates $\Delta G(V_\Delta, E_\Delta)$ on citation graphs, which is bounded by the minimum cover of affected node pairs K_{min} and the affected edges $\mathsf{AFFE_m}$ and $\mathsf{AFFE_s}$. Besides, the factor $O(|\mathsf{AFFE_m}| + |\mathsf{AFFE_s}|)$ in ours arises from the maintenance of the partition, which is not considered for these algorithms. Specifically, (1) modified AHRSZ, MNR, PK and PK$_2$ can detect and maintain SCCs. AHRSZ takes $O(|E_\Delta|\|\mathsf{AFF}\|log\|\mathsf{AFF}\|)$ time, and its $\|\mathsf{AFF}\|$ is larger than ours, as it is defined on G, instead of subgraphs. Further, the covers of affected node pairs in MNR, PK and PK$_2$ are also larger than our $\|\mathsf{AFF}\|$ [30] as they aim for simple solutions. (2) HKMST and BFGT are bounded by $|E|$, and incSCC$^+$ is bounded relative to Tarjan. However, these algorithms fail to capture the least amount of nodes to be visited when detecting SCCs, and their time complexities are incomparable with ours.

Algorithm incGm2Gm in our incremental methods is inspired by AHRSZ [2]. Differently, (1) incGm2Gm defines a cover of affected node pairs of edge insertions, and modifying procedure discover to find the cover bounded by $\|K_{min}\|$, (2) incGm2Gm only performs on the dummy nodes of subgraph G_m, while AHRSZ performs on the entire graphs and some unnecessary nodes and edges are visited, and (3) procedure maintain is simplified by removing the floor computation ($O(\|\mathsf{AFF}\|log\|\mathsf{AFF}\|)$ [2, 30]), and the number of newly created orders is relaxed to keep the total order of the nodes.

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

9 CONCLUSIONS

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

A couple of topics need a further study. One is to handle deletions for scholarly data updates although this is a rare case [12, 22, 33], and the other is to clean incorrect citations based on detected SCCs to improve the quality of the scholarly data analytic tasks [9, 23].

REFERENCES

[1] 2021. Aminer. https://www.aminer.cn/citation.

[2] Bowen Alpern, Roger Hoover, Barry K Rosen, Peter F Sweeney, and F Kenneth Zadeck. 1990. Incremental evaluation of computational circuits. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*. 32–42.

[3] Michael A Bender, Jeremy T Fineman, and Seth Gilbert. 2009. A new approach to incremental topological ordering. In *Proceedings of the twentieth annual ACM-SIAM symposium on Discrete algorithms*. SIAM, 1108–1115.

[4] Michael A Bender, Jeremy T Fineman, Seth Gilbert, Tsvi Kopelowitz, and Pablo Montes. 2017. File maintenance: when in doubt, change the layout!. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 1503–1522.

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

[6] Pavel Berkhin. 2005. A Survey on PageRank Computing. *Internet Mathematics* 2, 1 (2005), 73–120.

[7] Aaron Bernstein and Shiri Chechi. 2018. Incremental topological sort and cycle detection in expected total time. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 21–34.

[8] Sayan Bhattacharya and Janardhan Kulkarni. 2020. An improved algorithm for incremental cycle detection and topological ordering in sparse graphs. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2509–2521.

[9] Xu Chu, Ihab F Ilyas, Sanjay Krishnan, and Jiannan Wang. 2016. Data cleaning: Overview and emerging challenges. In *Proceedings of the 2016 international conference on management of data*. 2201–2206.

[10] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to algorithms*. MIT press.

[11] Paul Dietz and Daniel Sleator. 1987. Two algorithms for maintaining order in a list. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. 365–372.

[12] Wenfei Fan, Chunming Hu, and Chao Tian. 2017. Incremental graph computations: Doable and undoable. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 155–169.

[13] Wenfei Fan, Ruochun Jin, Muyang Liu, Ping Lu, Xiaojian Luo, Ruiqi Xu, Qiang Yin, Wenyuan Yu, and Jingren Zhou. 2020. Application driven graph partitioning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1765–1779.

[14] Wenfei Fan, Muyang Liu, Chao Tian, Ruiqi Xu, and Jingren Zhou. 2020. Incrementalization of graph partitioning algorithms. *Proceedings of the VLDB Endowment* 13, 8 (2020), 1261–1274.

[15] Harold N. Gabow. 2000. Path-based depth-first search for strong and biconnected components. *Inform. Process. Lett.* 74, 3 (2000), 107–114.

[16] Christos Giatsidis, Dimitrios M Thilikos, and Michalis Vazirgiannis. 2013. D-cores: measuring collaboration of directed graphs based on degeneracy. *Knowledge and information systems* 35, 2 (2013), 311–343.

[17] Bernhard Haeupler, Telikepalli Kavitha, Rogers Mathew, Siddhartha Sen, and Robert E Tarjan. 2012. Incremental cycle detection, topological ordering, and strong component maintenance. *ACM Transactions on Algorithms (TALG)* 8, 1 (2012), 1–33.

[18] Ilias Kanellos, Thanasis Vergoulis, Dimitris Sacharidis, Theodore Dalamagas, and Yannis Vassiliou. 2021. Impact-Based Ranking of Scientific Publications: A Survey and Experimental Evaluation. *IEEE Transactions on Knowledge and Data Engineering* 33, 4 (2021), 1567–1584.

[19] Ilias Kanellos, Thanasis Vergoulis, Dimitris Sacharidis, Theodore Dalamagas, and Yannis Vassiliou. 2021. Ranking papers by their short-term scientific impact. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1997–2002.

[20] Irit Katriel and Hans L Bodlaender. 2006. Online topological ordering. *ACM Transactions on Algorithms (TALG)* 2, 3 (2006), 364–379.

[21] Jungeun Kim and Jae-Gil Lee. 2015. Community detection in multi-layer graphs: A survey. *ACM SIGMOD Record* 44, 3 (2015), 37–48.

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

[23] Peng Li, Xi Rao, Jennifer Blase, Yue Zhang, Xu Chu, and Ce Zhang. 2021. CleanML: a study for evaluating the impact of data cleaning on ml classification tasks. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 13–24.

[24] Junfeng Liu, Shuai Ma, Renjun Hu, Chunming Hu, and Jinpeng Huai. 2020. Athena: A Ranking Enabled Scholarly Search System. In *Proceedings of the 13th International Conference on Web Search and Data Mining*. 841–844.

[25] Dongsheng Luo, Shuai Ma, Yaowei Yan, Chunming Hu, Xiang Zhang, and Jin-Peng Huai. 2022. A Collective Approach to Scholar Name Disambiguation. *IEEE Transactions on Knowledge and Data Engineering* 34, 5 (2022), 2020–2032.

[26] Shuai Ma, Chen Gong, Renjun Hu, Dongsheng Luo, Chunming Hu, and Jinpeng Huai. 2018. Query Independent Scholarly Article Ranking. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 953–964.

[27] Alberto Marchetti-Spaccamela, Umberto Nanni, and Hans Rohnert. 1996. Maintaining a topological order under edge insertions. *Inform. Process. Lett.* 59, 1 (1996), 53–58.

[28] Fabio Mercorio, Mario Mezzanzanica, Vincenzo Moscato, Antonio Picariello, and Giancarlo Sperli. 2021. DICO: A Graph-DB Framework for Community Detection on Big Scholarly Data. *IEEE Transactions on Emerging Topics in Computing* 9, 4 (2021), 1987–2003.

[29] David J Pearce. 2016. A space-efficient algorithm for finding strongly connected components. *Inform. Process. Lett.* 116, 1 (2016), 47–52.

[30] David J Pearce and Paul HJ Kelly. 2007. A dynamic topological sort algorithm for directed acyclic graphs. *Journal of Experimental Algorithmics (JEA)* 11 (2007), 1–7.

[31] David J Pearce and Paul HJ Kelly. 2010. A batch algorithm for maintaining a topological order. In *Proceedings of the Thirty-Third Australasian Conferenc on Computer Science-Volume 102*. 79–88.

[32] Dragomir R Radev, Pradeep Muthukrishnan, Vahed Qazvinian, and Amjad Abu-Jbara. 2013. The ACL anthology network corpus. *Language Resources and Evaluation* 47, 4 (2013), 919–944.

[33] Liam Roditty and Uri Zwick. 2008. Improved dynamic reachability algorithms for directed graphs. *SIAM J. Comput.* 37, 5 (2008), 1455–1471.

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

[35] Jeremy Siek, Lie-Quan Lee, Andrew Lumsdaine, et al. 2002. *The boost graph library*. Pearson India.

[36] Arnab Sinha, Zhihong Shen, Yang Song, Hao Ma, Darrin Eide, Bo-June (Paul) Hsu, and Kuansan Wang. 2015. An Overview of Microsoft Academic Service (MAS) and Applications. In *Proceedings of the 24th International Conference on World Wide Web*. 243–246.

[37] Jie Tang, Alvis C. M. Fong, Bo Wang, and Jing Zhang. 2012. A Unified Probabilistic Framework for Name Disambiguation in Digital Library. *IEEE Transactions on Knowledge and Data Engineering* 24, 6 (2012), 975–987.

[38] Jie Tang, Jing Zhang, Limin Yao, Juanzi Li, Li Zhang, and Zhong Su. 2008. ArnetMiner: Extraction and Mining of Academic Social Networks. In *KDD'08*. 990–998.

[39] Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM journal on computing* 1, 2 (1972), 146–160.

[40] Jian Wu, Kyle Mark Williams, Hung-Hsuan Chen, Madian Khabsa, Cornelia Caragea, Suppawong Tuarob, Alexander G. Ororbia, Douglas Jordan, Prasenjit Mitra, and C. Lee Giles. 2015. CiteSeerX: AI in a Digital Library Search Engine. *Ai Magazine* 36, 3 (2015), 35–48.

SUPPLEMENTARY MATERIAL

Appendix I: Real-life SCC Example

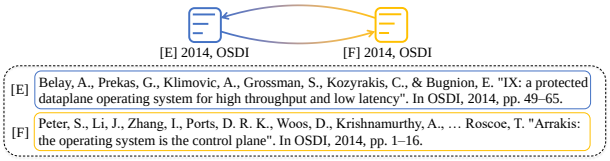


Figure 14: Another real-life SCC formed by the time gap

The SCC formed by both the time gap and scholarly systems wrongly parsing has illustrated in Example 1 in Section 1. 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 [1], where the nodes E and F represent two articles published at OSDI 2014. After the two articles are accepted, the committee finds that they study the same topic, and the authors are further asked to compare their systems with each other, which leads to the mutual citations and formation of an SCC. □

Appendix II: Static SCC Detection Algorithm

We present the detail of algorithm staDSCC in Section 3.2, which only traverses the nodes and edges involving within E_{s2s} and E_{o2n} to detect all SCCs, shown in Fig. 15.

Data structures & variables. staDSCC involves 4 data structures and 3 variables, *i.e.,* three local arrays inG_m, inG_{si}, $rindex$, global stack S, local variable $root$, two global variables $index$ and c. (1) Local array inG_m is a flag map indicating whether a node is visited by the heads of edges in E_{o2n}. (2) Local array inG_{si} maps a node to its associated *year* if it is visited by the heads of edges in E_{s2s}. (3) Local array $rindex$ maps each visited node to the visit order 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 order. After DFS is finished, $rindex$ maps each visited node to its component identifier. (4) Global stack S stores the nodes of the current SCC. (5) Local variable $root$ is used to identify the component root of an SCC. (6) Global variable $index$ indicates the order of the current visit node. (7) Global variable c is used to assign each SCC a non-increasing unique identifier, and all members of an SCC share the same c.

Procedure visit1 detects the SCCs involving within E_{o2n}. Given an unvisited head v in E_{o2n}, arrays $rindex$ and inG_m, it traverses G using DFS to find the SCCs reachable from v if exists. (1) $root$ and inG_m are set to $true$, $rindex[v]$ is initialized to $index$, and $index$ 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 order ($rindex[v] = min(rindex[v], rindex[w])$) (lines 5, 6), and v is pushed into S as v is not the component root ($root = false$) (line 12). (3) If $root = true$, the local root of a node is itself, and the traversal of the SCC reachable from v is finished. Then, all members of the SCC except $root$ are popped out from S, and assigned the same component identifier c (lines 7-11). To ensure $rindex[v] \leq rindex[w]$ for next w, which requires that $index < c$ always holds, c and $index$ are decreased by 1 (lines 8, 10, 11).

Input: Citation graph $G(V, E)$, E_{o2n}, E_{s2s}, max_year, min_year.
Output: Each node's component identifier $rindex$, inG_m and inG_{si}.

1. Initialize $rindex[v] = 0$, $inG_m[v] = false$, $inG_{si}[v] = 0$ for all $v \in V$;
2. $index = 1$; $c = |V| - 1$; $S = \emptyset$;
3. **for each** edge $e \in E_{o2n}$
4. **if** $rindex[e.head] = 0$ **then** visit1$(e.head, rindex, inG_m)$;
5. **for each** i from max_year **to** min_year **do**
6. **for each** edge $e \in E_{s2si}$
7. **if** $rindex[e.head] = 0$ **then** visit2$(e.head, i, rindex, inG_{si})$;
8. **return** $rindex$, inG_m, inG_{si}.

Procedure visit1$(v, rindex, inG_m)$

1. $root = true$; $inG_m[v] = true$;
2. $rindex[v] = index$; $index = index + 1$;
3. **for each** $(v, w) \in E$ **do**
4. **if** $rindex[w] = 0$ **then** visit1$(w, rindex, inG_m)$;
5. **if** $rindex[w] < rindex[v]$ **then**
6. $rindex[v] = rindex[w]$; $root = false$;
7. **if** $root$ **then**
8. $index = index - 1$;
9. **while** $S \neq \emptyset$ **and** $rindex[v] \leq rindex[S.top()]$ **do**
10. $w = S.pop()$; $rindex[w] = c$; $index = index - 1$;
11. $rindex[v] = c$; $c = c - 1$;
12. **else** $S.push(v)$.

Procedure visit2$(v, i, rindex, inG_{si})$

1. $root = true$; $inG_{si}[v] = i$;
2. $rindex[v] = index$; $index = index + 1$;
3. **for each** $(v, w) \in E$ **and** $w.year = i$ **do**
4. **if** $rindex[w] = 0$ **then** visit2$(w, i, rindex, inG_{si})$;
5. **if** $rindex[w] < rindex[v]$ **then**
6. $rindex[v] = rindex[w]$; $root = false$;
7. **if** $root$ **then**
8. $index = index - 1$;
9. **while** $S \neq \emptyset$ **and** $rindex[v] \leq rindex[S.top()]$ **do**
10. $w = S.pop()$; $rindex[w] = c$; $index = index - 1$;
11. $rindex[v] = c$; $c = c - 1$;
12. **else** $S.push(v)$.

Figure 15: Algorithm staDSCC

Procedure visit2 detects the SCCs involving within E_{s2s}, which is a slight variant of visit1. Specifically, given an unvisited head v in E_{s2si}, its *year* i, arrays $rindex$ and inG_{si}, it only traverses the unvisited nodes of the same-year subgraph to find the SCCs. Same as procedure visit1, visit2 also identifies an SCC by finding its component root and sets $rindex$ to its component identifier. There are two differences between procedures visit1 and visit2. (1) The array inG_{si} is set to its *year* i (line 1). (2) For each unvisited successor w of v, it restricts the year of w to i (line 3). Hence, the nodes not in *year* i and visited by visit1 are both pruned in 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 node component identifier $rindex$, arrays inG_m and inG_{si}. First, inG_m, inG_{si} and $rindex$ are initialized to $false$, 0 and 0 for each node v, $index$ is set to 1, c is set to $|V| - 1$, and S is initialized to an empty stack, respectively (lines 1, 2). It then calls procedure visit1 to detect all SCCs reachable from the heads of E_{o2n} for each unvisited head v (lines 3, 4). 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 5-7). Finally, it returns *rindex*, inG_m, inG_{si} such that the set of SCCs of G can be assembled by iterating *rindex* and gathering the nodes by the component identifier (line 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.

Appendix III: Proofs on Partitioning

We present the detailed proofs of Propositions in Section 4.1 on partitioning citation graphs.

Proof of Proposition 4: From Property 1 in Section 3.1, 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 5: First, algorithm staDSCC traverses from all edge heads of E_{o2n} on the entire graph via Pearce [29], hence all nodes and edges of G_m are found by staDSCC. Second, for each year i, it traverses from the edge heads of E_{s2si} on the unvisited nodes of same-year subgraphs via Pearce, hence all nodes and edges of G_{si} are also found by staDSCC. Third, G_r and E_c can be easily obtained if G_m and each G_{si} is found. Note that, inG_m and inG_{si} of staDSCC are utilized to represent the partition. □

Appendix IV: Proofs on Local Topological Order

We present the detailed proof of the Proposition in Section 4.3 on the local topological order.

Proof of Proposition 9: We show this by combining the properties of Pearce [29]. As illustrated, we have generated the DAG representation of the graph after SCCs are detected. First, the topological order of the DAG is also a byproduct of Pearce. 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. That is the *rindex* of each node can also be utilized to represent the topological order of the DAG representation of the graph. Second, all nodes and edges of G_m and G_{si} are visited by algorithm staDSCC, which is designed on the basis of Pearce without changing its logic. Thus, staDSCC produces the topological orders of the DAG representations of G_m and each G_{si}, which means staDSCC produces the local topological order of G. □

Appendix V: Correctness Proof of incGm2Gm

We present the detailed proof of Theorem 12 in Section 5.2 on incremental algorithm incGm2Gm.

Proof of Theorem 12: It is trivial for (x, y) in the same SCC or with a valid local topological order. We consider (x, y) violates the local topological order, and show this from (1) AFF found by algorithm incGm2Gm is a cover of affected node pairs, (2) $\|\mathsf{AFF}\| \leq 2\|K_{min}\|$, where K_{min} is the minimum cover of affected node pairs, (3) algorithm incGm2Gm correctly detects the SCCs if exists, and (4) it correctly maintains the partition and local topological order. Note that, we only consider the DAG representation of G_m, and dx, dy are the dummy nodes of x, y, respectively.

(1) We first show AFF is a cover of affected node pairs. Let $V_f = \{s | dy \rightsquigarrow s \wedge (ord(s) < ord(dx) \vee s = dx)\}$, and $V_b = \{s | s \rightsquigarrow dx \wedge (ord(s) > ord(dy) \vee s = dy)\}$, where $dy \rightsquigarrow s$ or $s \rightsquigarrow dx$ is a path 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 $ord(s) < ord(t)$ 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 $\forall s \in V_b - \mathsf{AFF}_b$ and $\forall t \in V_f - \mathsf{AFF}_f$, then $ord(s) < ord(t)$ holds.

Let s_h be the node in $V_b - \mathsf{AFF}_b$ with the highest topological order, and t_l be the node in $V_f - \mathsf{AFF}_f$ with the lowest topological order. Based on the termination condition of procedure discover (line 4), $ord(t_l) > ord(s_h)$ holds. Hence, for any $s \in V_b - \mathsf{AFF}_b$ and any $t \in V_f - \mathsf{AFF}_f$, $ord(s) < ord(s_h) \vee s = s_h$ and $ord(t_l) < ord(t) \vee t_l = t$ always holds, which means $ord(s) < ord(t)$.

(2) We then show $\|\mathsf{AFF}\| \leq 2\|K_{min}\|$, where K_{min} is the minimum cover of affected node pairs. AFF is also divided into AFF_f and AFF_b. Actually, either AFF_f or AFF_b is the minimum cover of affected node pairs, as for $\forall t \in \mathsf{AFF}_f$, $\forall s \in \mathsf{AFF}_b$, pairs (s, t) are 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 $\forall t \in \mathsf{AFF}_f$, $ord(s) > ord(t) \vee s = t$ holds for all paths $s \rightsquigarrow t$, *i.e.,* pairs (s, t) are 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 $\|\mathsf{AFF}_f\| = \|\mathsf{AFF}_b\|$, and AFF_f or AFF_b is the minimum cover. Hence, $\|\mathsf{AFF}\| = \|\mathsf{AFF}_f\| + \|\mathsf{AFF}_b\| = 2\|\mathsf{AFF}_b\| = 2\|\mathsf{AFF}_f\| = 2\|K_{min}\|$. In short, $\|\mathsf{AFF}\| \leq 2\|K_{min}\|$ always holds.

(3) We next show incGm2Gm correctly detects the SCCs if exists. Assume there exists a new SCC after the edge insertion (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. If (x, y) introduces a new SCC, then x and y both belong to the SCC, and it is easy to perform DFS only on the nodes of AFF to detect all members of the 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 6. (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. In incGm2Gm, rTO is the reverse local topological orders of the dummy nodes in the connected subgraph induced by AFF based on [10]. Following the topological orders of nodes in AFF, we can easily create an order for each node s lower than its $Ceiling[s]$ by insertBefore function of the ordered list. Hence, incGm2Gm correctly maintains the topological order of G_m.

Putting these together, we have proved Theorem 12. □

Appendix VI: Correctness Proof of incGm2Gsr

We present the detailed proof of Theorem 13 in Section 5.3 on incremental algorithm incGm2Gsr.

Proof of Theorem 13: 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 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 inG_m 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 6.

(2) When maintaining the topological order of G_m, there are three types of edges violating the topological order on the dummy nodes 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 $(s,t) \in E_1$, $ord(s) < ord(t)$ is easily maintained, as s and t are found by scanGsGr1 (DFS based algorithm) which also produces their topological orders [10]. For each node $s \in V_{dsi} \cup V_{dr}$, $t \in V_{dm}$, v_{lo} is the node with the lowest order of t. It creates an order for each s lower than v_{lo}, hence $ord(s) < ord(t)$ holds for each $(s,t) \in E_2$. For (dx, dy) of E_3, $ord(dx) < ord(dy)$ is guaranteed by algorithm incGm2Gm.

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

Note that, algorithm incGm2Gm finds a cover AFF of affected node pairs, and AFF is bounded such that $\|\mathsf{AFF}\| \leq 2\|K_{min}\|$, where K_{min} is the minimum cover of affected node pairs.

Putting these together, we have proved Theorem 13. □

Appendix VII: Correctness Proof of incGsr2Gsr

We present the detailed proof of Theorem 14 in Section 5.4 on incremental algorithm incGsr2Gsr.

Proof of Theorem 14: We prove the correctness of algorithm incGsr2Gsr from three cases (1) $(x, y) \in E_{n2o}$, (2) $(x, y) \in E_{o2n}$, (3) $(x, y) \in E_{s2s}$. 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 dummy nodes of V_m, V_{si} and V_r, respectively.

(1) For $(x, y) \in E_{n2o}$, the partition and local topological order are valid, and no SCC is introduced by Propositions 4, 6, and 10.

(2) For $(x, y) \in E_{o2n}$, (a) the partition of G_m, G_{si} and G_r can be easily maintained with the same analysis as Theorem 13. (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 dummy nodes 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 13, for each edge $(s,t) \in E_1 \cup E_2$, $ord(s) < ord(t)$ holds as procedure scanGsGr2 is designed by slightly revising procedure scanGsGr1.

(3) For $(x, y) \in E_{s2s}$, it is considered as follows.
(i) $x \in G_{si}$, $y \in G_{si}$. (a) The partition is valid by Proposition 6. Algorithm incGsr2Gsr correctly (b) maintains the local topological order, and (c) detects the SCC if exists using algorithm incGm2Gm on the dummy nodes of subgraph G_{si} by Theorem 12.
(ii) $x \in G_r$, $y \in G_{si}$. The partition and local topological order are valid, and no SCC is introduced based on Propositions 4, 6, and 10.
(iii) $x \in G_r$, $y \in G_r$. (a) Algorithm incGsr2Gsr marks y in G_{si}, then the partition is valid based on Proposition 7. (b) The local topological order is easily maintained by creating dy an order lower than its out-neighbors in subgraph G_{si} in algorithm incGsr2Gsr. This is because only the edges in $E = \{(dy, t)|t \in V_{dsi}\}$ violate the topological order. (c) No extra SCCs are introduced as the nodes in G_{si} do not reach to dx based on the definition of G_r.
(iv) $x \in G_{si}$, $y \in G_r$. Algorithm incGsr2Gsr correctly (a) maintains the partition, (b) maintains the local topological order, and (c) detects the SCC if exists by combining the analyses of (iii) and (i).

Note that, algorithm incGm2Gm finds a cover AFF of affected node pairs, and AFF is bounded such that $\|\mathsf{AFF}\| \leq 2\|K_{min}\|$, where K_{min} is the minimum cover of affected node pairs.

Putting these together, we have proved Theorem 14. □

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 5 & 6.

(1) Time complexity of single update algorithm incGm2Gm**.** The time cost arises from procedure 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, $O(\|\mathsf{AFF}_b\|)$ 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 memSCCs 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, 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} as $\|\mathsf{AFF}\| \leq 2\|K_{min}\|$, which takes the least amount of work when the out or in-neighbors of the cover are required to be traversed.

(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

Table 5: Space cost of static algorithms (MB)

Datasets	G	Pearce	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

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

(3) Time complexity of single update algorithm incGsr2Gsr**.** The time cost arises from (a) the maintenance of partition, (b) the maintenance of local topological order and the SCC detection.

(a) It takes $O(|\mathsf{AFFE_m}| + |\mathsf{AFFE_{si}}|)$ time to maintain the partition. (i) If $(x, y) \in E_{o2n}$, it takes also $O(|\mathsf{AFFE_m}|)$ time using scanGsGr2, with the same analysis as incGm2Gsr. (ii) If $(x, y) \in E_{s2s}, x, y \in G_r$, it takes $O(|\mathsf{AFFE_{si}}|)$ time. Before inserting (x, y), and after handling (x, y) by incGsr2Gsr, the partition of G are both valid. All edges starting from y and reaching to G_{si} are the edges that need to be added to G_{si}, *i.e.,* $\mathsf{AFFE_{si}}$. Algorithm incGm2Gsr only traverses the out-neighbors of dy on the dummy nodes G_{si} in a one-pass manner, which actually traverses $|\mathsf{AFFE_{si}}|$ edges and visits less than $|\mathsf{AFFE_{si}}|$ nodes. (iii) If $(x, y) \in E_{s2s}, x \in G_{si}, y \in G_r$, it also takes $O(|\mathsf{AFFE_{si}}|)$ time, with the same analysis as (ii). For the other cases of algorithm 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) If $(x, y) \in E_{o2n}$, it takes $O(|\mathsf{AFFE_m}|)$ time as scanGsGr2 detects the SCC and maintains the local topological order. (ii) If $(x, y) \in E_{s2s}, x, y \in G_r$, it takes $O(|\mathsf{AFFE_{si}}|)$ time to only maintain the local topological order, as it only traverses the out-neighbors of dy on the dummy nodes G_{si} in a one-pass manner. (iii) If $(x, y) \in E_{s2s}, x, y \in G_{si}$, incGm2Gm takes $O(\|\mathsf{AFF}\|log\|\mathsf{AFF}\|)$ time on G_{si}. (iv) If $(x, y) \in E_{s2s}, x \in G_{si}$, $y \in G_r$, it takes $O(|\mathsf{AFFE_{si}}| + \|\mathsf{AFF}\|log\|\mathsf{AFF}\|)$ time by combining (ii) and (iii). For the other cases of algorithm incGsr2Gsr, the local topological order remains valid and no extra SCCs are introduced.

From these, algorithm incGsr2Gsr takes $O(\|\mathsf{AFF}\|log\|\mathsf{AFF}\| + |\mathsf{AFFE_m}| + |\mathsf{AFFE_s}|)$ time, where (1) AFF is a cover of the affected node pairs, (2) $\mathsf{AFFE_m}$ is the affected edges (*i.e.,* those added edges) on G_m, and (3) $\mathsf{AFFE_s}$ is the affected edges (*i.e.,* those added edges) on G_s, after inserting edge (x, y) with $x \in G_s \cup G_r$ and $y \in G_s \cup G_r$. Note that, incGsr2Gsr is bounded by the minimum cover of affected node pairs K_{min} as $\|\mathsf{AFF}\| \leq 2\|K_{min}\|$, the affected edges $\mathsf{AFFE_m}$ 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, five shared data structures, *i.e.,* arrays inG_m, inG_{si}, memSCCs, ordered lists oLists, disjoint set dSet, and arrays inF, inB in algorithm incGm2Gm, arrays Rdy, $isVisit$ in algorithm incGm2Gsr, and array $inSCC$ in algorithm incGsr2Gsr. The storage of the citation graph costs $O(|V| + |E|)$ space. Each of arrays inG_m, inG_{si} and memSCCs 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, sinDSCC takes $O(|V| + |E|)$ space for single updates.

(5) Time complexity of batch update algorithm batDSCC**.** The time cost 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 $O(|E_\Delta|\|\mathsf{AFF}\|log\|\mathsf{AFF}\| + |\mathsf{AFFE_m}| + |\mathsf{AFFE_s}|)$ 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.

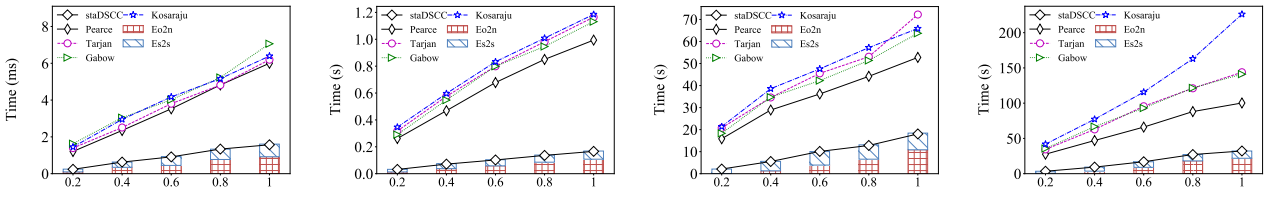
From these, algorithm batDSCC takes $O(|E_\Delta|\|\mathsf{AFF}\|log\|\mathsf{AFF}\| + |\mathsf{AFFE_m}|+|\mathsf{AFFE_s}|+|V_\Delta|)$ time for batch updates $\Delta G(V_\Delta, E_\Delta)$, where (1) AFF is bounded, *i.e.,* $\|\mathsf{AFF}\| \leq 2\|K_{min}\|$ such that K_{min} is the minimum cover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

Appendix IX: Detailed and Extra Tests of Static Algorithm

We present the detailed and extra static algorithm tests in Section 7 and we compare the running time and space cost of staDSCC with its competitors. The impacts of $|E_{o2n}|$ and $|E_{s2s}|$ on the efficiency of static algorithms are also evaluated.

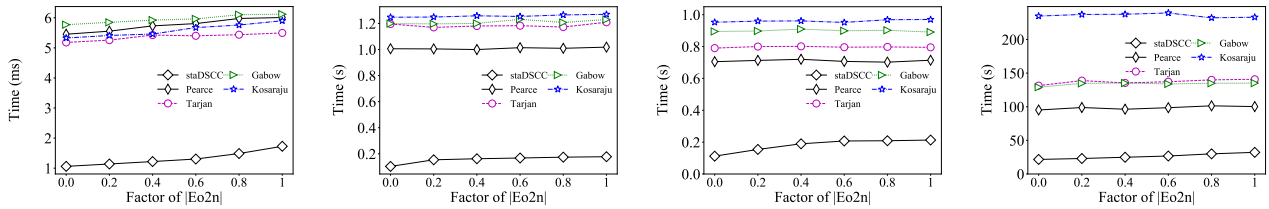
(1) Running time tests. To evaluate the impacts of the graph size, we vary $|G|$ with the scale factors from 0.2 to 1, where $|E|/|V|$ is kept fixed. The results are reported in Fig. 16.

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 Pearce, 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 Pearce, 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%,

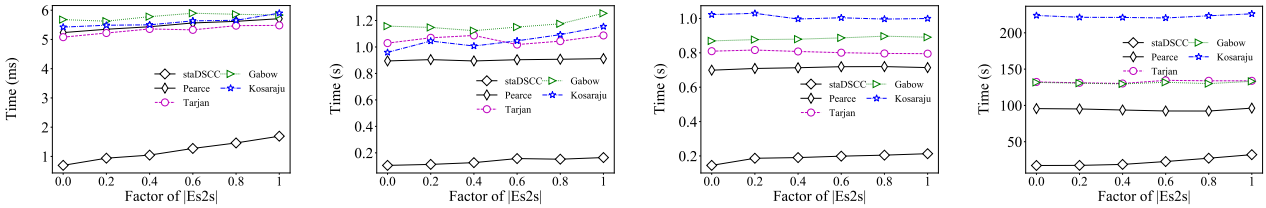


(a) AAN (b) DBLP (c) ACM (d) MAG

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}

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 in 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 5.

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 Pearce that reduces the space requirements, and staDSCC avoids unnecessary visits of nodes and edges.

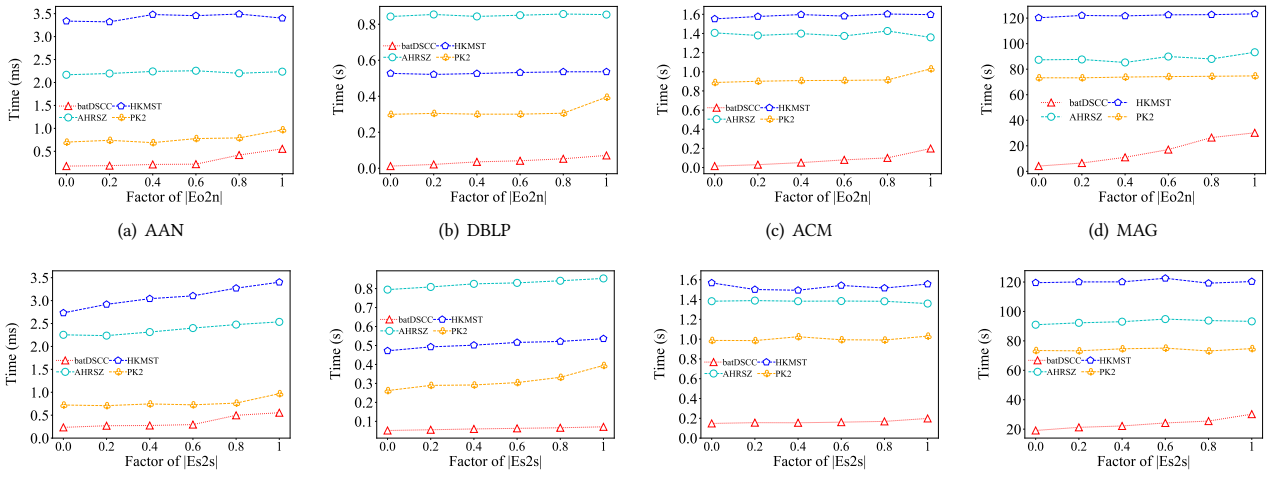
(3) The efficiency of static algorithms *w.r.t.* the number of E_{o2n} and E_{s2s}. We separately study the impacts of $|E_{o2n}|$ and $|E_{s2s}|$ on the efficiency of static algorithms, *i.e.,* staDSCC, Pearce, Tarjan, Gabow and Kosaraju. The results are reported in Fig. 17.

To evaluate the impacts of $|E_{o2n}|$, we vary $|E_{o2n}|$ with a factor from 0 to 1 on the entire datasets (AAN, DBLP, ACM, MAG), 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 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(b), 17(c) & 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% in four citation graphs. (2) Traversing edges in E_{o2n} could find a large of common nodes and edges when traversing edges in E_{s2s}, which leads to the slow increase of $|V_{ns}|$ and $|E_{ns}|$ in staDSCC. Note that it has a time complexity of $\Theta(|V_{ns}| + |E_{ns}|)$.

To evaluate the impacts of $|E_{s2s}|$, we vary $|E_{s2s}|$ with a factor from 0 to 1 on the entire datasets (AAN, DBLP, ACM, MAG), where $|E_{o2n}|$ and $|E_{n2o}|$ are kept fixed, *i.e.,* only a factor of edges of E_{s2s} are kept (all the other edges of E_{s2s} are deleted), and all edges of E_{o2n} and E_{n2o} remain untouched in citation graphs. The results are reported in Figs. 17(e), 17(f), 17(g) & 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.



(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 6: The number of edges performing graph traversals of batch updates

Datasets	Number of edge insertions	AHRSZ	HKMST	incSCC$^+$	MNR	PK$_2$	sinDSCC	batDSCC
		Number of invalid weak topological order edges	Number of invalid topological order edges				Number of invalid partition or local topological order 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

Table 7: 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

Appendix X: Extra Tests of Incremental Algorithm

We present the extra incremental algorithm tests in Section 7. We first evaluate the impacts of $|E_{o2n}|$ and $|E_{s2s}|$ on the efficiency of incremental algorithms. Then, the affected nodes of single updates and the number of edges involved with graph traversals of batch updates are also studied.

(1) 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 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(b), 18(c) & 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 $|\mathsf{AFFE_m}|$ and $\|\mathsf{AFF}\|$. Note that it has a time complexity of $O(|\mathsf{AFFE_m}| + |\mathsf{AFFE_s}| + |V_\Delta| + |E_\Delta|\|\mathsf{AFF}\|log\|\mathsf{AFF}\|)$.

To evaluate the impacts of $|E_{s2s}|$, we vary $|E_{s2s}|$ with a factor from 0 to 1 on both G and ΔG of (AAN, DBLP, ACM, MAG), where $|E_{o2n}|$ and $|E_{n2o}|$ are kept fixed and $|\Delta G| = 10\%$, *i.e.,* only a factor of edges of E_{s2s} are kept (the other edges of E_{s2s} are 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(f), 18(g) & 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 $\|\mathsf{AFF}\|$.

(2) Affected nodes of single updates. For single updates of randomly inserting 15,000 edges, we summarize the average affected nodes per edge insertion of algorithms sinDSCC, AHRSZ, HKMST, incSCC$^+$, MNR and PK$_2$ in Table 7. All algorithms require to maintain the orders for their affected nodes. We find that the average affected nodes per edge insertion found by sinDSCC are on average (8.5, 7.7, 11670.8, 1156576.6, 1156741.1) times less than AHRSZ, HKMST, incSCC$^+$, MNR and PK$_2$ on four real-life citation graphs.

Note that, incSCC$^+$, MNR and PK$_2$ are designed based on DFS which leads to traversing more affected nodes.

(3) The number of edges performing graph traversals of batch updates. For batch updates $|\Delta G| = 30\%$, we summarize the number of edges performing graph traversals of algorithms batDSCC, AHRSZ, HKMST, incSCC$^+$, MNR and PK$_2$ in Table 6. Note that, AHRSZ is designed based on the weak topological order, and HKMST, incSCC$^+$, MNR, PK$_2$ are all designed based on the topological order. Our sinDSCC and batDSCC are based on the partition and local topological order. Algorithms perform graph traversals for edges with invalid (weak) topological order or invalid partition and local topological order.

We find the following. First, sinDSCC and batDSCC both reduce at least (87%, 94%, 92%, 95%) invalid edges than its best incremental competitors on (AAN, DBLP, ACM, MAG), respectively. Second, batDSCC further reduces (5%, 23%, 26%, 2%) invalid edges than sinDSCC on (AAN, DBLP, ACM, MAG), respectively. Third, once incremental SCC detection for scholarly data is designed based on topological order or its variants, most inserted edges (>83.82%) are valid, *i.e.,* the graph traversal is unnecessary. To conclude, sinDSCC and batDSCC capture the properties of citation graphs, and batDSCC further reduces the amount of graph traversals, which is consistent with the analyses in Sections 4, 5 & 6.