Incremental SCC Maintenance in Sparse Graphs

Aaron Bernstein
Rutgers University, New Brunswick, NJ, USA

Aditi Dudeja
Rutgers University, New Brunswick, NJ, USA

Seth Pettie
University of Michigan, Ann Arbor, MI, USA

Abstract

In the *incremental cycle detection* problem, edges are added to a directed graph (initially empty), and the algorithm has to report the presence of the first cycle, once it is formed. A closely related problem is the *incremental topological sort* problem, where edges are added to an acyclic graph, and the algorithm is required to maintain a valid topological ordering. Since these problems arise naturally in many applications such as scheduling tasks, pointer analysis, and circuit evaluation, they have been studied extensively in the last three decades. Motivated by the fact that in many of these applications, the presence of a cycle is not fatal, we study a generalization of these problems, *incremental maintenance of strongly connected components* (incremental SCC).

Several incremental algorithms in the literature which do cycle detection and topological sort in directed acyclic graphs, such as those by [7] and [16], also generalize to maintain strongly connected components and their topological sort in general directed graphs. The algorithms of [16] and [7] have a total update time of $O(m^{3/2})$ and $O(m \cdot \min\{m^{1/2}, n^{2/3}\})$ respectively, and this is the state of the art for incremental SCC. But the most recent algorithms for incremental cycle detection and topological sort ([8] and [10]), which yield total (randomized) update time $\tilde{O}(\min\{m^{4/3}, n^2\})$, do not extend to incremental SCC. Thus, there is a gap between the best known algorithms for these two closely related problems.

In this paper, we bridge this gap by extending the framework of [10] to general directed graphs. More concretely, we give a Las Vegas algorithm for incremental SCCs with an expected total update time of $\tilde{O}(m^{4/3})$. A key ingredient in the algorithm of [10] is a structural theorem (first introduced in [8]) that bounds the number of "equivalent" vertices. Unfortunately, this theorem only applies to DAGs. We show a natural way to extend this structural theorem to general directed graphs, and along the way we develop a significantly simpler and more intuitive proof of this theorem.

2012 ACM Subject Classification Theory of computation → Dynamic graph algorithms

Keywords and phrases Directed Graphs, Strongly Connected Components, Dynamic Graph Algorithms

Digital Object Identifier 10.4230/LIPIcs.ESA.2021.14

Funding *Aaron Bernstein*: This work was done while funded by NSF Award 1942010.

1 Introduction

In dynamic algorithms, our main goal is to maintain a key property of the graph while an adversary makes changes in the graph in the form of edge insertions and deletions. An algorithm is called incremental if it handles only insertions, decremental if it handles only deletions and fully dynamic if it handles both insertions as well as deletions. For a dynamic algorithm we hope to optimize the update time of the algorithm, which is the time taken by the algorithm to adapt to the changes to the input and modify the results. For incremental/decremental algorithms, one typically seeks to minimize the *total* update time over the entire sequence of edge insertions/deletions.

In this paper, we consider the problem of maintaining strongly connected components in the incremental setting (incremental SCC). This is a generalization of the problems of incremental cycle detection and topological sorting in directed acyclic graph, which find application in pointer analysis, deadlock detection [4], circuit evaluation [3] and scheduling tasks. In many of these applications, the presence of a cycle is not fatal, which motivates the general problem of maintaining strongly connected components, as well as the topological order of these components.

The problems of incremental cycle detection and topological sorting were first studied by Katriel and Bodlaender [18], who gave the first non-trivial algorithm for these problems with a total update time of $O(\min\{m^{3/2}\log n, m^{3/2} + n^2\log n\})$. This bound was improved by Liu and Chao [20] to $O(m^{3/2} + m\sqrt{n}\log n)$. Since then, these problems and incremental SCC have been studied extensively (see for example [1, 2, 6, 16, 7, 13, 21]). Several algorithms that do incremental cycle detection and topological sort maintenance in directed acyclic graphs can be modified to get algorithms for incremental SCC. For example, the algorithm of Haeupler, Kavitha, Mathew, Sen and Tarjan [16] is able to do cycle detection as well as strongly connected component maintenance in $O(m^{3/2})$ total update time. In an important result, Bender, Fineman, Gilbert and Tarjan presented two algorithms for strongly connected components, with total update times of $O(n^2\log n)$ and $O(m\cdot\min\{m^{1/2}, n^{2/3}\})$, for dense and sparse graphs, respectively (see Table 1).

The two most recent algorithms in this area are limited to cycle detection and topological sort: Bernstein and Chechik [8] gave a Las Vegas algorithm with an expected total update time of $O(m\sqrt{n}\log n)$; Bhattacharya and Kulkarni [10] combined the balanced search approach of [16] with the results of [8] to get an algorithm with a total expected runtime of $\tilde{O}(m^{4/3})$. As a result, there was still a gap between the best known algorithms for cycle detection and topological sort (update time of $\tilde{O}(\min\{m^{4/3}, n^2\})$ and for incremental SCC (update time of $\tilde{O}(\min\{m^{3/2}, n^2\})$). In this paper, we bridge the gap between these closely related problems. More formally, we prove the following result.

► **Theorem 1.** *There exists an incremental algorithm for maintaining strongly connected components in directed graphs with expected total time $\tilde{O}(m^{4/3})$, where m refers to the number of edges in the final graph. The algorithm can also maintain the topological order of these components.*

Summary of Techniques. We obtain our results by extending the technique of [10] to the case of general directed graphs. Both [8] and [10] detect cycles by doing a graph search after the insertion of an edge (u, v). However, they reduce their search space by only exploring "equivalent" vertices: vertices whose ancestor and descendant sets agree on a random subset S of V. A key ingredient of the analysis is a structural theorem of [8] that bounds the total number of equivalent pairs created by the sequence of insertions. However, their notion of equivalent vertices only applies to acyclic directed graphs. Additionally, the proof of this structural theorem (Lemma 3.2 and 3.5 of [8]) is rather unintuitive.

Our contributions are three-fold. We present a new proof of the structural theorem of [8], which is significantly simpler and more intuitive. We also show a natural generalization of this theorem to general directed graphs. Finally, we show how the framework of [10] can be extended to maintain SCCs in general graphs, rather than just doing cycle detection and topological sort in a DAG.

Related Problems. A closely related problem that has received a lot of attention is maintaining strongly connected components in a *decremental* graph. This problem has been widely studied (see e.g. [22, 19, 11, 9]) and a recent algorithm achieves near-optimal $\tilde{O}(m)$

Table 1 Known Results for Incremental Cycle Detection, Topological Sort and SCC.

Reference	Update Time	Incremental SCC
[18]	$O(\min\{m^{3/2}\log n, m^{3/2} + n^2 \log n\})$	No
[20]	$O(m^{3/2} + m\sqrt{n}\log n)$	No
[2]	$O(n^{2.75})$	No
[6]	$\tilde{O}(n^2)$	No
[16]	$O(m^{3/2})$	Yes
[7]	$O(m \cdot \min\{m^{1/2}, n^{2/3}\}), \tilde{O}(n^2)$	Yes
[8]	$\tilde{O}(m\sqrt{n})$	No
[10]	$\tilde{O}(m^{4/3})$	No

total expected update time [9]. Although the goal in both problems is to maintain SCCs, the incremental and decremental versions have little overlap in terms of techniques. Another related problem is that of maintaining single-source shortest paths in an incremental directed graph. The current state-of-the-art for this problem is $\tilde{O}(n^2)$ in dense graphs [15] and $\tilde{O}(m\sqrt{n} + m^{7/5})$ in sparse ones [12].

2 Preliminaries

We consider the problem of maintaining strongly connected components in directed graphs in the incremental setting. In this setting, we start with an empty graph, and directed edges are added to the graph one at a time. We will let G refer to the current version of the graph, and its vertex and edge sets are denoted as V and E respectively. We use m to denote the total number of edges added to G and n to denote $|V(G)|$.

Consider two vertices $u, v \in V$. We say that the vertex u is an *ancestor* of v, and v is a *descendant* of u if there is a path from u to v in G. We will say that u and v are *related* if one is the ancestor of other. For $u \in V$, we use $A(u)$ and $D(u)$ to denote the current set of ancestors and descendants of u. Consider any $S \subseteq V$, for $u \in V$, we use $A_S(u)$ to denote the set $A(u) \cap S$, and $D_S(u)$ to denote the set $D(u) \cap S$. For any $v \in V$, we will use $C(v)$ to denote the strongly connected component containing v in the current graph G, and $|C(v)|$ will be the number of vertices contained in the component.

We will also use the following result due to Italiano [17] on single-source incremental reachability.

▶ **Lemma 2** ([17]). *Given $v \in V$, there exists an algorithm that maintains $A(v)$ and $D(v)$ in $O(m)$ total time during the course of insertion of m edges.*

We also use the following simplifying assumption by [8] (proved in the appendix of their paper).

▶ **Lemma 3** ([8]). *We can assume that every vertex in the current graph $G = (V, E)$ has degree $O(m/n)$.*

Data Structures Used. To maintain the strongly connected components, we use the disjoint set data structure of Tarjan [23]. This data structure stores the partition of the vertex set into disjoint sets. In our case, these disjoint sets will be the strongly connected components. Moreover, the disjoint sets are represented by a *canonical element*, which in this case will be a vertex. Following operations are supported by this data structure.

1. Find(x): Given a vertex x, return the canonical vertex of the component containing x.
2. Link(x, y): This operation joins the components whose canonical vertices are x and y. The newly formed component's canonical vertex is x.

This data structure supports any sequence of Find and Link operations in $O(n \log n)$ total time plus $O(1)$ time per operation. Our search and reordering operations will take $\Omega(n \log n)$ total time, so we can think of the Find and Link operations as being performed in $O(1)$ amortized time per operation.

Additionally, to maintain the topological ordering of the strongly connected component, we use the *ordered list* data structure of [14] and [5], which supports the following operations in $O(1)$-time.

1. Insert-Before(x, y): This operation inserts the vertex x before the vertex y in the ordered list.
2. Insert-After(x, y): This operation inserts the vertex x after the vertex y in the ordered list.
3. Delete(x): This operation deletes the vertex x from the current ordered list.
4. Order(x, y): This operation returns whether x appears before y in the ordering or not.

This data structure maintains the topological sort k of the strongly connected components implicitly. We will use some additional data structures for our algorithm, that we will mention when we discuss the algorithm.

3 Similarity

3.1 Previous Work

To bound the running time of their algorithm [8] introduced the notion of *sometime-τ-similar* pairs. We briefly discuss their definition.

▶ **Definition 4** ([8]). *A pair of vertices u and v are said to be sometime-τ-similar if there is a time t at which u is an ancestor of v, $|A(u) \oplus A(v)| \leq \tau$, and $|D(u) \oplus D(v)| \leq \tau$.*

The total number of *sometime-τ-similar* pairs are $\tilde{O}(n\tau)$. Note that this bound is false if we apply the same definition of similarity to the case of directed graphs with cycles. As an example, consider the case where the entire graph is a cycle. For such a graph, by Definition 4, we have $O(n^2)$ *sometime-τ-similar* pairs. So, a new definition of similarity is needed. Moreover, their proof strategy also uses the final topological ordering of the graph. Such an ordering is not possible in directed graphs with cycles. We overcome this by defining another ordering that (like topological ordering) is consistent with the incremental updates to the graph, but at the same time allows for strongly connected components.

3.2 A New Notion Of Similarity

▶ **Definition 5.** *Consider $u, v \in V$. Let $C(u)$ and $C(v)$ denote the strongly connected components containing u and v respectively, then u and v are called* τ-similar *in the current graph G if u and v are related, $|C(u)| \leq \tau$, $|C(v)| \leq \tau$, and $|A(u) \oplus A(v)| \leq \tau$, $|D(u) \oplus D(v)| \leq \tau$. Vertices u and v are called* sometime-τ-similar*, if they are* τ-similar *at some point during the course of m edge insertions.*

▶ Remark 6. Consider any $u, v \in V$ with $C(u) = C(v)$. If $|C(u)| \geq \tau + 1$ then u and v are not τ-similar in G. But if $C(u) \leq \tau$ then they are τ-similar.

With this remark, we distinguish between two types *sometime-τ-similar* vertices.

▶ **Definition 7.** *We call u and v* related-sometime-τ-similar *if there is a time t when u and v are τ-similar with $C(u) \neq C(v)$. On the other hand if there is a time t when u and v are τ-similar and $C(u) = C(v)$, then we call u and v* equivalent-sometime-τ-similar. *It is possible for u, v to be both* related-sometime-τ-similar *and* equivalent-sometime-τ-similar.

We show that the total number of *sometime-τ-similar* pairs are bounded.

▶ **Theorem 8.** *The total number of* sometime-τ-similar *pairs are $\tilde{O}(n\tau)$.*

Our proof will bound *related-sometime-τ-similar* pairs. It is easy to see that the number of *equivalent-sometime-τ-similar* is $O(n\tau)$.

▶ **Observation 9.** *A vertex v can only be* equivalent-sometime-τ-similar *to the first τ vertices that join the same component as v. Thus, the total number of* equivalent-sometime-τ-similar *pairs is $O(n\tau)$.*

To prove Theorem 8 we need the following claim.

▷ Claim 10. There exists a fixed total order I on the vertices of G which satisfies the following property:

1. Consider any $u, v \in V$. Let t_1 be the first time u and v become related such that u is an ancestor of v, then $I(u) < I(v)$.

Note that if the final graph G_m is acyclic, then I is satisfied by the topological ordering. We will show that it is possible to obtain an ordering that satisfies the above properties even if the graph has a cycle.

3.3 Existence of A Fixed Total Order

In this subsection, we define an ordering I that satisfies Claim 10.

▶ **Definition 11.** *We define a relation $\prec$ over the vertices of G: $u \prec v$ if and only if at some time t, u is an ancestor of v and $C(u) \neq C(v)$.*

We first note that $\prec$ is a strict partial order. We formally state and prove the following claim.

▷ Claim 12. The relation $\prec$ on the vertices of G is a strict partial order.

Proof. We need to show that $\prec$ is anti-symmetric and transitive. Anti-symmetry follows from the fact that for each pair of vertices u and v, either $u \not\prec v$ or $v \not\prec u$. Now suppose $u \prec v$ and $v \prec w$. Let t_1 be the time at which u is an ancestor of v and $C(u) \neq C(v)$. Similarly, let t_2 be the time at which v is an ancestor of w and $C(v) \neq C(w)$. Without loss of generality, assume that $t_2 \geq t_1$. Observe that u is an ancestor of w at time t_2. If $C(u) = C(w)$, then $v \in C(w)$ at time t_2 as well, which is a contradiction. So, at time t_2, $C(u) \neq C(w)$. This proves our claim. ◁

▶ **Definition 13.** *We define I to be a linear extension of $\prec$. That is I is a total order consistent with $\prec$: if $u \prec v$, then $I(u) < I(v)$.*

Proof of Claim 10. We claim that I of Definition 13 satisfies Claim 10. Consider any two vertices u and v, and let t_1 be the time at which u and v first become related, with u being an ancestor of v. Therefore at time t_1, $C(u) \neq C(v)$, which implies that $u \prec v$. Since I is consistent with $\prec$, we know that $I(u) < I(v)$. ◁

3.4 Bounding the Number of Similar Pairs

In this section we will prove Theorem 8. From Observation 9, we conclude that it is sufficient to show that the number of *related-sometime-τ-similar* pairs are at most $O(n\tau \log n)$. We first introduce some notation. Moving forward we will use I to denote an ordering that satisfies Claim 10. We note that Theorem 8 can be obtained by combining the ordering I satisfying Claim 10 with a modification of the proof of *sometime-τ-similar* pairs in a DAG in Section 3 of [8]. However, even for the simpler case of DAGs, the proof in [8] requires a long case analysis. In this paper we present a different approach to the proof we believe is significantly simpler and more intuitive.

▶ **Definition 14.** *Let u and v be a pair of* related-sometime-τ-similar *vertices. We denote it using an ordered tuple (u, v) if $I(u) < I(v)$.*

▶ **Definition 15.** *For a vertex v, we define $A^i(v)$ to be the set of vertices u such that (u, v) is a* related-sometime-τ-similar *pair, and $I(v) - I(u) \in [2^i, 2^{i+1})$. Similarly, we define $D^i(v)$ to be the set of vertices w such that (v, w) is a* related-sometime-τ-similar *pair, and $I(w) - I(v) \in [2^i, 2^{i+1})$.*

▶ **Definition 16.** *For a vertex v and a fixed i, we define the graph $G_v^{D,i}$ with the vertex set $D^i(v)$ and the graph $G_v^{A,i}$ with the vertex set $A^i(v)$ as follows.*

1. *Let $u_1, u_2, \cdots, u_\alpha$ be the vertices of $A^i(v)$, where the vertices are ordered according to the increasing order of the time at which they become* related-τ-similar *with v. For $j < k$, we add an edge from u_j to u_k, if u_j is an ancestor of u_k when u_k first becomes τ-similar to v.*
2. *Let $w_1, w_2, \cdots, w_\beta$ be the vertices of $D^i(v)$, where the vertices are ordered according to the increasing order of time at which they become* related-τ-similar *with v. For $j < k$, we add an edge from w_j to w_k if w_j is a descendant of w_k when w_k first becomes τ-similar to v.*

See Figure 1 for an illustration.

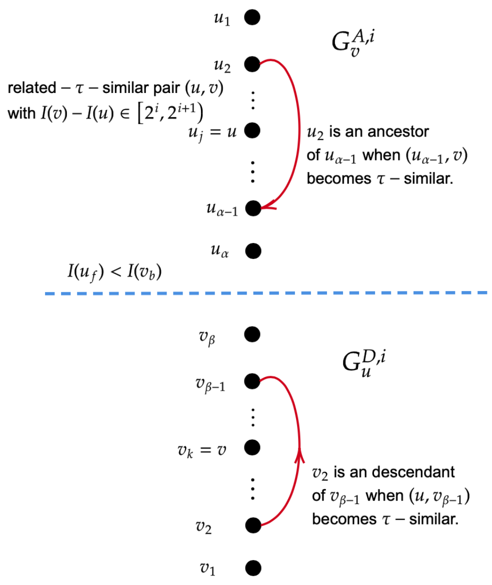
▷ Claim 17. Let (u, v) be a *related-τ-similar* pair such that $I(v) - I(u) \in [2^i, 2^{i+1})$. Consider $w \in A^i(v)$ and $z \in D^i(u)$, then $I(w) < I(z)$.

Proof. Suppose $I(z) < I(w)$. Note that $I(u) < I(z)$, and $I(w) < I(v)$. Consequently, $I(u) < I(z) < I(w) < I(v)$. Since $I(z) - I(u) \geq 2^i$, and $I(v) - I(w) \geq 2^i$, this implies that $I(v) - I(u) \geq 2^{i+1}$, which contradicts our assumption that $I(v) - I(u) \in [2^i, 2^{i+1})$. ◁

▷ Claim 18. For a vertex v, consider any $A^i(v) = \{u_1, \cdots, u_\alpha\}$, where u_j are ordered in the increasing order of time at which they become *related-τ-similar* to v. Then the number of edges in $G_v^{A,i}$ coming into u_j is at least $j - \tau$. Similarly, let $D^i(v) = \{w_1, \cdots, w_\beta\}$, where the vertices are ordered in the increasing order of time at which they become *related-τ-similar* with v. Then the number of edges coming into w_j in $G_v^{D,i}$ is at least $j - \tau$.

Proof. Let t be the time at which (u_j, v) become *related-τ-similar.* By t, for all $i < j$, (u_i, v) are *related-τ-similar.* If the in-degree of u_j is at most $j - \tau - 1$, then this implies that there are at least $\tau + 1$ vertices u_i, $i < j$ such that u_i is not an ancestor of u_j. However, these are all ancestors of v at time t. This implies that $|A(u_j) \oplus A(v)| \geq \tau + 1$, contradicting the fact that u_j and v are *related-τ-similar* at time t. ◁

▶ **Definition 19.** *For a vertex v, consider $w \in A^i(v)$. We call w* bad *with respect to v if the outdegree of w in $G_v^{A,i}$ is at most 2τ. Similarly, we call a vertex $z \in D^i(v)$* bad *with respect to v if the outdegree of z in $G_v^{D,i}$ is at most 2τ.*



u_1
$G_v^{A,i}$
u_2
related $- \tau -$ similar pair (u, v)
with $I(v) - I(u) \in [2^i, 2^{i+1})$
$u_j = u$
u_2 is an ancestor
of $u_{\alpha-1}$ when $(u_{\alpha-1}, v)$
becomes $\tau -$ similar.
$u_{\alpha-1}$
u_α
$I(u_f) < I(v_b)$
v_β
$G_u^{D,i}$
$v_{\beta-1}$
$v_k = v$
v_2 is an descendant
of $v_{\beta-1}$ when $(u, v_{\beta-1})$
becomes $\tau -$ similar.
v_2
v_1

Figure 1 We consider a *related-τ-similar* pair (u, v), where $I(v) - I(u) \in [2^i, 2^{i+1})$. All vertices of $A^i(v)$ appear before the vertices of $D^i(u)$.

▷ Claim 20. For any v, the total number of bad vertices in $A^i(v)$ for any i is at most 6τ. Similarly, the total number of bad vertices in $D^i(v)$ for any i is at most 6τ.

Proof. As before, let $A^i(v) = \{u_1, u_2, \cdots, u_\alpha\}$. Let $B = \{u_{\alpha-4\tau+1}, \cdots, u_\alpha\}$. Let $A \subset A^i(v) \setminus B$ be the set of vertices outside of B that are bad for v (see Figure 2 for an illustration). We want to prove that $|A| \leq 2\tau$. This will give us the desired bound. Consider any $w \in B$. There are at least $|A| - \tau$ edges from A to w. So, the total number of edges going from A to B is at least $4\tau(|A| - \tau)$. The average outdegree of the vertices in A is at least $\frac{4\tau(|A|-\tau)}{|A|}$. Since the vertices in A are bad, we know that $\frac{4\tau(|A|-\tau)}{|A|} \leq 2\tau$. This implies that $|A| \leq 2\tau$. The proof for $D^i(v)$ is analogous. ◁

▶ **Lemma 21.** *Let (u, v) be a* related-sometime-τ-similar *pair. Then, either u is bad for v or v is bad for u.*

Proof. Let $I(v) - I(u) \in [2^i, 2^{i+1})$. As before we consider $A^i(v) = \{u_1, \cdots, u_\alpha\}$, and let $D^i(u) = \{v_1, \cdots, v_\beta\}$. Assume that neither u is bad for $A^i(v)$ nor v is bad for $D^i(u)$. This implies that the number of edges going out of u and v in $G_v^{A,i}$ and $G_u^{D,i}$, respectively, are at least $2\tau + 1$. Consider the *related-τ-similar* pairs $(u_1, v), \cdots, (u_\alpha, v)$ and $(u, v_1), \cdots, (u, v_\beta)$. Note that among these pairs one of (u_α, v) or (u, v_β) are the last to become *related-τ-similar*. Without loss of generality, assume it is (u, v_β). Since we assume that u is not bad with respect to v, at the point when (u_α, v) becomes *related-τ-similar*, u is an ancestor of at least $2\tau + 1$ vertices in $u_1, \cdots, u_\alpha$. Note that this claim also holds at the (later) time when (u, v_β) become *related-sometime-τ-similar*. We call this set of vertices U. We now consider two different cases:

1. If v_β is not an ancestor of at least $\tau + 1$ vertices in U, then this contradicts the fact that (u, v_β) is a *related-τ-similar* pair.
2. Suppose v_β is an ancestor of at least $\tau + 1$ vertices in U. Consider any $u_j \in U$. Observe from Claim 17 that $I(u_j) < I(v_\beta)$. Since I satisfies Claim 10, we deduce that if v_β is an

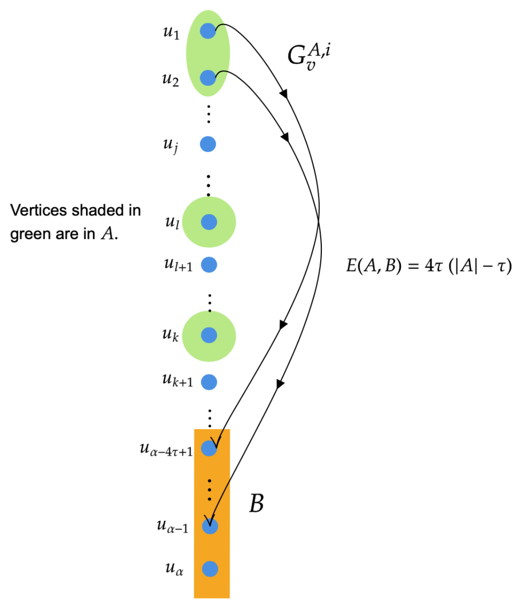


Figure 2 The vertices in green are the vertices of $A^i(v) \setminus B$ that are bad for v. Since the vertices in B are τ-similar to v, the total number of edges coming out of A and going into B is at least $4\tau(|A| - \tau)$.

ancestor of u_j, then it lies in the same strongly connected component as u_j. Since this is true for all $u_j \in U$, it follows that $|C(v_\beta)| \geq \tau + 1$, thus contradicting the fact that (u, v_β) is *related-τ-similar* (see Definition 5). ◀

▶ Remark 22. Observe that when we are in the acyclic case, then we don't have to deal with the second case at all, since I corresponds to the topological ordering of the final graph G_m.

Proof of Theorem 8. Consider a *related-τ-similar* pair (u, v). We charge this pair to u if v is bad for u, and we charge it to v if u is bad for v. From Lemma 21, we know that each pair (u, v) is charged to either u or v. Finally, we observe that for any u, for a fixed i, the total number of bad vertices in $D^i(u)$ or $A^i(u)$ is at most 6τ each. Therefore, the total charge on each vertex is at most $12\tau \log n$ (since i is at most $\log n$). Since the total number of vertices is n, we know that the total charge, and therefore the total number of *related-sometime-τ-similar* pairs is at most $O(n\tau \log n)$. ◀

4 Equivalence

Consider $u, v \in V$, observe that u and v lie in the same strongly connected component iff $A(u) = A(v)$ and $D(u) = D(v)$. However, the sets $A(u)$ and $D(v)$ are expensive to maintain for all vertices. Therefore, Bernstein and Chechik [8] defined a relaxed notion of equivalence between vertices. We define a slightly different version that will be useful for our algorithm.

▶ **Definition 23.** *(S-equivalence) Consider S which is created by including every vertex $v \in V$ independently with probability $^{12 \cdot \log n}/_{\tau}$, where τ is a parameter to be defined by the algorithm. Vertices u and v are called S-equivalent if they are related, $A_S(u) = A_S(v)$, and $D_S(u) = D_S(v)$. For the analysis of our algorithm, it will be useful to distinguish between two types of S-equivalence.*

1. *Vertices x and y are* Type 1 *if they satisfy the above-mentioned condition and $|C(x)| \geq \tau+1$ or $|C(y)| \geq \tau + 1$.*
2. *Vertices x and y are* Type 2 *if they satisfy the above-mentioned condition and $|C(x)| \leq \tau$ and $|C(y)| \leq \tau$.*

In [8], the algorithm samples a set S, and maintains a partition $\{V_{i,j}\}$ of V, where $V_{i,j} = \{u \in V \text{ s.t. } |A_S(u)| = i, |D_S(u)| = j\}$. We define an ordering $\prec^*$ on these parts.

▶ **Definition 24.** *We say that $V_{i,j} \prec^* V_{k,l}$ if either $\{i < k\}$, or $\{i = k$ and $j > l\}$ (note the slightly unusual ordering, instead of $j < l$, we have $j > l$). For $x \in V$, we use $V(x)$ to denote partition $V_{i,j}$ that contains x.*

This partition has the following properties which were proved in [8] for directed acyclic graphs, but extend to general directed graphs as well.

▶ **Lemma 25.** *Let $\{V_{i,j}\}$ be the partition of V maintained by the algorithm determined by the sampled set S, then*
1. *If x and y are related, with x being an ancestor of y, then either $V(x) \prec^* V(y)$ or $V(x) = V(y)$.*
2. *Consider a strongly connected component C of the current graph G, then $C \subseteq V_{i,j}$ for some i, j.*
3. *If x and y are related, and $V(x) = V(y)$, then x and y are S-equivalent.*

Proof. We give a short proof of this lemma. If x is an ancestor of y, then $A(x) \subseteq A(y)$, and $D(y) \subseteq D(x)$. In particular, $A_S(x) \subseteq A_S(y)$ and $D_S(y) \subseteq D_S(x)$. This immediately tells us that $|A_S(x)| \leq |A_S(y)|$, and $|D_S(y)| \leq |D_S(x)|$ and the first part of the claim follows. Finally, consider any strongly connected component C, then for any $x, y \in C$, $A(x) = A(y)$, $D(x) = D(y)$. This implies that $A_S(x) = A_S(y)$, $D_S(x) = D_S(y)$ and this proves the second part of the claim. To see the third part, assume without loss of generality that x is an ancestor of y, and $V(x) = V(y)$. Note that $A_S(x) \subseteq A_S(y)$, and since $|A_S(x)| = |A_S(y)|$, we can conclude that $A_S(x) = A_S(y)$. Similarly, we can deduce that $D_S(x) = D_S(y)$, thus proving that x and y are S-equivalent. ◀

Keeping in mind Lemma 25, for a component C, we define $V(C)$ as the partition $V_{i,j}$ containing C. An important component of our algorithm is maintaining a topological sort k of the strongly connected components. This topological sort k will be consistent with the order $\prec^*$ of the partitions. That is, for strongly connected components C and C' with $V(C) \prec^* V(C')$, $k(C) < k(C')$. The existence of such a topological ordering is guaranteed by Lemma 25. We will maintain a topological sort of the components by maintaining an ordered list on the canonical vertices. The components are disjoint and each of them have a unique canonical vertex. So, we will often use $k(\cdot)$ on canonical vertices as well.

The algorithms in [8] and [10] proceed by exploiting the notion of S-equivalence. This notion enables them reduce the space of vertices that need to be explored to detect cycles (from Lemma 25). Finally, they show that with high probability the total number of S-equivalent pairs is bounded, and the runtime of the algorithm is proportional to this number. In order to prove this claim, they show that S-equivalent pairs and sometime-τ-similar pairs are related. We show that our notion of sometime-τ-similarity can be used to bound the number of Type 2 S-equivalent pairs, instead of all S-equivalent pairs. It will be clear as we move forward why this is sufficient.

Recall Definition 23 and Definition 5. We show the following lemma relating *sometime-τ-similar* pairs and *Type 2* S-equivalent pairs.

▶ **Lemma 26.** *Suppose $S \subseteq V$ is obtained by including each $x \in V$ independently with probability $\frac{12 \cdot \log n}{\tau}$. Suppose u and v are* Type 2 *S-equivalent, then with high probability, they are* sometime-τ-similar.

Observe that Theorem 8 and Lemma 26 together imply the following theorem:

▶ **Theorem 27.** *Let S be sampled by including $v \in V$ independently with probability $\frac{12 \cdot \log n}{\tau}$. Then, the total number of Type 2 S-equivalent pairs is at most $\tilde{O}(n\tau)$ with high probability.*

We now proceed to prove Lemma 26.

Proof of Lemma 26. Note that by the statement of the lemma, u and v are related, $|C(u)| \leq \tau$, $|C(v)| \leq \tau$. We additionally want to show that $|D(u) \oplus D(v)| \leq \tau$ and $|A(u) \oplus A(v)| \leq \tau$. Without loss of generality, assume that $|A(u) \oplus A(v)| \geq \tau + 1$. Then, applying Chernoff bound, we conclude that with probability at least $1 - O(1/n^5)$ there is a vertex $x \in A(u) \oplus A(v)$ that is included in S as well. This implies that u and v are not Type 2 S-equivalent. Taking union bound over all Type 2 S-equivalent pairs, which are at most n^2 in number, we conclude that with probability at least $1 - O(1/n^3)$ any Type 2 S-equivalent pair is also sometime-τ-similar. ◀

5 The Algorithm

When an edge (u, v) is inserted, the algorithm updates the newly formed strongly connected components, if any. Additionally, the algorithm maintains a topological sort k of the strongly connected components. This will be achieved by using canonical vertices as a proxy for the strongly connected components (see Section 2). These canonical vertices will be maintained as an ordered list, and when we are required to reorder the strongly connected components, the corresponding canonical vertices will be reordered. To achieve this, we follow the basic framework of [10]. The algorithm to process the insertion of (u, v) proceeds in the following phases.

1. **Phase 1.** This phase is responsible for maintaining reachability information to and from S (using Lemma 2). Additionally, in this phase, the algorithm uses this reachability information to update the sets $V_{i,j}$ and to handle the case where the new SCC formed by the insertion of (u, v) contains at least one vertex in S. If the algorithm finds such an SCC, it terminates after Phase 1, i.e. it skips Phases 2 and 3.
2. **Phase 2.** This phase is responsible for handling small SCCs. In particular, it detects the case when (u, v) creates a new SCC that does not contain any $s \in S$, as well as the case where (u, v) creates no new SCC. The phase also links together the canonical vertices corresponding to this new SCC (if any).
3. **Phase 3.** This phase updates the topological order of the strongly connected components by reordering canonical vertices. Note that even if (u, v) creates no SCCs, Phase 3 may need to do some reordering to ensure that k remains a valid topological order.

In the main body of the paper, we will describe Phases 2 and 3 and the subroutines used in these phases. The correctness of these subroutines can be found in the full version of the paper. Phase 1 is essentially the same as in the framework of [8], so we postpone the details to the full version of the paper.

5.1 Phase 1: Updating the partition $\{V_{i,j}\}$ and Handling Large SCCs

In this section, we give an overview of Phase 1 and its guarantees. The detailed description is in the full version of the paper. Using Lemma 2, we can maintain reachability to and from every vertex in S in total time $O(m|S|)$ over all edge insertions. This allows us to maintain two additional piece of information. Recall the partition $V_{i,j}$ from Section 4, and note that every $V(x)$ is determined entirely by $A_S(x)$ and $D_S(x)$. Thus, Phase 1 can use the reachability information to/from S to maintain the partition $V_{i,j}$. Phase 1 can also use this reachability information to detect any new SCCs that contain a vertex in S.

We now state these guarantees more formally. The following lemma is essentially identical to the guarantees of [8], but is modified to handle SCCs.

▶ **Lemma 28** ([8])**.** *Consider the insertion of edge (u, v). Phase 1 has the following guarantees:*

1. *At the end of Phase 1, each set $V_{i,j}$ is correct for the new version of the graph (the graph with edge (u, v) inserted). The algorithm also updates the order of the strongly connected components so that they are consistent with $\prec^*$.*
2. *If the insertion of (u, v) creates a new SCC that contains a vertex in S, then Phase 1 detects the new SCC, links the corresponding canonical vertices, and computes the topological order of the resulting SCCs. The update procedure then terminates and does not continue to Phase 2 or 3.*
3. *If the insertion of (u, v) does not create a new SCC that contains a vertex in S, then Phase 1 does not create any new SCCs. In this case, after the end of Phase 1, the ordering k on the canonical vertices is guaranteed to be a valid topological ordering of the canonical vertices in $G \setminus \{(u, v)\}$. The algorithm then proceeds to Phases 2 and 3.*

5.2 Phase 2: Detecting Small SCCs

The algorithm enters Phase 2 only if the newly inserted edge (u, v) does not create a new SCC that contains a vertex of S; otherwise the algorithm to process (u, v) terminates after Phase 1. We also remark that if the algorithm enters Phase 2, then with high probability the size of the newly formed strongly connected component (if one exists) is at most τ. This follows from an easy application of Chernoff bound: if the newly formed component has size at least $\tau + 1$, then with high probability, it contains a vertex of S, in which case the algorithm terminates after Phase 1. Taking a union bound over all n^2 edge insertions, we get the following:

▶ **Observation 29.** *If the algorithm enters Phase 2 while processing an edge (u, v), then with high probability, the new strongly connected components formed by the addition of (u, v) (if one exists) has size at most τ.*

Additionally, recall that Phase 1 updates the partition set $\{V_{i,j}\}$, so we assume that once we enter Phase 2 this partition already corresponds to the graph G (Lemma 28).

Previous Work. Our Phase 2 will be similar to the cycle detection algorithm of [10], but we need to adapt it to find the newly formed strongly connected component. Previous algorithms for finding SCCs such as the one by [16], proceed by implementing the cycle detection algorithm, but running it only over the canonical vertices. However, our algorithm will do a search over all vertices of the graph. We do this because sizes of the SCCs will be relevant to the runtime of the algorithm, and they weren't relevant in the case of [16].

We now give a brief outline of Phase 2: when an edge (u, v) is added to the graph, then the algorithm first checks if $k(\textsc{Find}(u)) < k(\textsc{Find}(v))$. If this is the case, then there couldn't have been an existing path from v to u (due to Lemma 28). As a result, a new component containing u and v could not be formed. So, the algorithm doesn't continue. To detect if a new component is formed and to find all the vertices of this component, the algorithm does alternate steps of forward and backward search. For this purpose, it maintains sets F_a and F_d (to do forward search), B_a and B_d (to do backward search). For the forward search, F_a and F_d are the vertices that are alive (yet to be explored), and dead (already explored). Sets B_a and B_d are similarly defined for the backward search. When we encounter a vertex while exploring in the forward direction, we add it to F_a. When all neighbors of a vertex $v \in F_a$ that are S-equivalent to v been added to $F_a \cup F_d$, we add v to F_d. We add vertices to B_a and B_d similarly. At all times, while exploring vertices in the forward direction, we want to stay as close to v as possible, so we pick out a vertex x with minimum $k(\textsc{Find}(x))$ from F_a to explore next. Similarly, while exploring in the backward direction, we want to stay as close to u as possible, so we pick out the vertex y with maximum $k(\textsc{Find}(y))$ from B_a to explore next in the backward direction. We refer the reader to Algorithms 1, 2, and 4 for pseudocodes for Phase 2. We give the proof of correctness in the full version of the paper. We briefly outline the proof of runtime.

▶ **Lemma 30.** *The total runtime of Phase 2 is* $O(\sqrt{m^3\tau/n})$.

Proof Sketch. Suppose we have process edge e_t and let f_t denote the size of F_d after $\textsc{FindComponent}()$ has finished terminated. We observe that $|B_d| = \Theta(f_t)$ as well, since we do a balanced search. From Lemma 3 we conclude that the total update time of the algorithm over m edge insertions is $O(m/n \sum_{t=1}^{m} f_t)$. The goal is to now bound $\sum_{t=1}^{m} f_t$. Consider $x \in F_d$ and $y \in B_d$ after $\textsc{FindComponent}()$ has finished processing e_t. We show that (x, y) is a newly formed *related-τ-similar* pair or *equivalent-τ-similar* pair. This implies that $\sum_{t=1}^{m} f_t^2 = \tilde{O}(n\tau)$ (from Theorem 8). Using Cauchy-Schwarz, we know that $\sum_{t=1}^{m} f_t = \tilde{O}(\sqrt{mn\tau})$. Thus the total runtime of Phase 2 is $O(\sqrt{m^3\tau/n})$. ◀

5.3 Phase 3: Sorting the Canonical Vertices

We enter this Phase only if there is no vertex of S in the newly created SCC, C_N. After Phase 1 and Phase 2, we know which canonical vertices have combined to give the newly formed strongly connected component. We delete these canonical vertices from the ordered list, and show how to reorder the list so that a topological sort on the canonical vertices is maintained.

To update the topological ordering of the canonical vertices, we follow the framework of [10]. We present it here for completeness, modifying their algorithm slightly to account for the case where a cycle is created.

We will consider two cases, one where a new component is created and one where no new component is created. Suppose no new component is created, and consider the sets F_d and B_d, from the forward and backward searches after we have processed edge (u, v). Since we do an ordered search, we know that all the vertices of a given component appear in a continuous manner in F_d and B_d. Let $\textsc{Find}(v), x_1, \cdots, x_f$ be the **canonical** vertices corresponding to the components appearing in F_d, with $k(\textsc{Find}(v)) < k(x_1) < \cdots < k(x_f)$. Similarly, let $y_1, y_2, \cdots, y_b, \textsc{Find}(u)$ be the **canonical** vertices corresponding to the components appearing in B_d, with $k(y_1) < k(y_2) < \cdots < k(y_b) < k(\textsc{Find}(u))$. We use the subroutine $\textsc{UpdateForward}()$ and $\textsc{UpdateBackward}()$ to update the ordered list (see Algorithm 3). This list only consists of canonical vertices that represent different components.

We now describe how to reorder the vertices. In [10] two cases are considered, the first case corresponds to when the algorithm terminates in conditions: $B_a = \emptyset$ or $\max_{x \in B_a} k(\textsc{Find}(x)) < \max_{y \in F_d} k(\textsc{Find}(y))$. For this case, we use the subroutine $\textsc{UpdateForward}()$. The proof for the case when the algorithm terminates in conditions $F_a = \emptyset$ or $\min_{x \in F_a} k(\text{FIND}(x)) > \min_{y \in B_d} k(\text{FIND}(y))$ is analogous (we use a subroutine UPDATEBACKWARD()) and we omit it here. We postpone the proof of correctness to the full version. We give a proof of the runtime.

▶ **Lemma 31.** *The total runtime of Phase 3 is $O(\sqrt{mn\tau})$.*

Proof sketch. For each $x \in F_d$ and $y \in B_d$, the algorithm UPDATEFORWARD() puts FIND(x) and FIND(y) in the correct position in the ordered list. This takes time $O(1)$ per vertex in F_d and B_d, giving a total runtime of $O(\sqrt{mn\tau})$. ◀

When a new component C_N is formed. If a new strongly connected component C_N is formed, and it doesn't contain a vertex of S, then the algorithm still needs to reorder some components. Assume without loss of generality that v is the canonical vertex of C_N. We first proceed to delete from the ordered list, all canonical vertices corresponding to the components that combined to form C_N. We define $x_1, x_2, \cdots x_f \in F_d$ and $y_1, y_2, \cdots, y_b \in B_d$ as before except we exclude the canonical vertices that combined to form C_N. Finally, if our FINDCOMPONENT() terminated in $B_a = \emptyset$ or $\max_{x \in B_a} k(\text{FIND}(x)) \leq \max_{y \in F_d} k(\text{FIND}(y))$, then we execute UPDATEFORWARD(), else if FINDCOMPONENT() terminated in $F_a = \emptyset$ or $\min_{x \in F_a} k(\text{FIND}(x)) \geq \min_{y \in B_d} k(\text{FIND}(y))$, then we execute UPDATEBACKWARD(). The proof of correctness can be found in the full version of the paper and is the same as in the case when there is no new strongly connected component formed.

▶ **Lemma 32.** *The total update time of our algorithm is $\tilde{O}(m^{4/3})$.*

Proof. The total time taken in Phase 1, 2 and 3 is at most $\tilde{O}(mn/\tau + \sqrt{mn\tau} + \sqrt{m^3\tau/n})$. Substituting $\tau = n/m^{1/3}$, we get the desired bound of $\tilde{O}(m^{4/3})$. ◀

Algorithm 1 EXPLORE-FORWARD(x).

```
$F_a = F_a \setminus \{x\}$ and $F_d = F_d \cup \{x\}$.
for $x' \in \text{out}(x)$ with $V(x) = V(x')$ do
    if FIND$(x') \in F_a \cup F_d$ then
        CYCLE $= 1$
    if $x' \notin B_a \cup B_d$ then
        add $x'$ to $B_a$.
```

Algorithm 2 EXPLORE-BACKWARD(x).

```
$B_a = B_a \setminus \{x\}$ and $B_d = B_d \cup \{x\}$.
for $x' \in \text{in}(x)$ with $V(x) = V(x')$ do
    if FIND$(x') \in F_a \cup F_d$ then
        CYCLE $= 1$
    if $x' \notin B_a \cup B_d$ then
        add $x'$ to $B_a$.
```

Algorithm 3 UPDATEFORWARD().

```
$Q = F_d$.
$x^* = \arg\max\{k(\textsc{Find}(x)) \mid x \in Q, x \text{ canonical}\}$.
$Q = Q \setminus C(x^*)$. // Since we are rearranging canonical vertices.
while $Q \neq \emptyset$ do
    $x' = \arg\max_{x \in Q}\{k(\textsc{Find}(x))\}$.
    $Q = Q \setminus C(x')$.
    $\textsc{Insert-Before}(\textsc{Find}(x'), x^*)$
    $x^* = \textsc{Find}(x')$.
$y^* = \textsc{Find}(v)$.
$Q = B_d$.
while $Q \neq \emptyset$ do
    $y' = \arg\max_{y \in B_d} k(\textsc{Find}(y))$.
    $Q = Q \setminus C(y')$.
    $\textsc{Insert-Before}(\textsc{Find}(y'), y^*)$
    $y^* = \textsc{Find}(y')$.
```

References

1 Deepak Ajwani and Tobias Friedrich. Average-case analysis of incremental topological ordering. *Discrete Appl. Math.*, 158(4):240–250, 2010. doi:10.1016/j.dam.2009.07.006.

2 Deepak Ajwani, Tobias Friedrich, and Ulrich Meyer. An o(n2.75) algorithm for incremental topological ordering. *ACM Trans. Algorithms*, 4(4), 2008. doi:10.1145/1383369.1383370.

3 Bowen Alpern, Roger Hoover, Barry K. Rosen, Peter F. Sweeney, and F. Kenneth Zadeck. Incremental evaluation of computational circuits. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '90, page 32–42, USA, 1990. Society for Industrial and Applied Mathematics.

4 Ferenc Belik. An efficient deadlock avoidance technique. *IEEE Trans. Comput.*, 39(7):882–888, 1990. doi:10.1109/12.55690.

5 Michael A. Bender, Richard Cole, Erik D. Demaine, Martin Farach-Colton, and Jack Zito. Two simplified algorithms for maintaining order in a list. In *Proceedings of the 10th Annual European Symposium on Algorithms*, ESA '02, page 152–164, Berlin, Heidelberg, 2002. Springer-Verlag.

6 Michael A. Bender, Jeremy T. Fineman, and Seth Gilbert. A new approach to incremental topological ordering. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '09, page 1108–1115, USA, 2009. Society for Industrial and Applied Mathematics.

7 Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Robert E. Tarjan. A new approach to incremental cycle detection and related problems. *ACM Trans. Algorithms*, 12(2):14:1–14:22, 2016. doi:10.1145/2756553.

8 Aaron Bernstein and Shiri Chechik. Incremental topological sort and cycle detection in $\tilde{O}(m\sqrt{n})$ expected total time. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 21–34. SIAM, 2018.

9 Aaron Bernstein, Maximilian Probst, and Christian Wulff-Nilsen. Decremental strongly-connected components and single-source reachability in near-linear time. In Moses Charikar and Edith Cohen, editors, *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, pages 365–376. ACM, 2019.

10 Sayan Bhattacharya and Janardhan Kulkarni. 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*, pages 2509–2521. SIAM, 2020.

Algorithm 4 FindComponent(u, v).

```
1  if $k(\textsc{Find}(u)) < k(\textsc{Find}(v))$ then
2      return NO.
3  Initialize CYCLE $= 0$ and min-heaps $F_a = \{v\}, B_a = \{u\}, F_d = \emptyset, B_d = \emptyset$.
4  if $\textsc{Find}(u) = \textsc{Find}(v)$ or $V(u) \neq V(v)$ then
5      return NO.
6  while $F_a \neq \emptyset$ and $B_a \neq \emptyset$ do
7      Let $x = \arg\min_{x' \in F_a} k(\textsc{Find}(x'))$.
8      if $k(\textsc{Find}(x)) > \min_{z' \in B_d} k(\textsc{Find}(z'))$ and CYCLE $= 0$ then
9          EXIT LOOP.
10     if $k(\textsc{Find}(x)) > \min_{z' \in B_d} k(\textsc{Find}(z'))$ and CYCLE $= 1$ then
11         EXIT LOOP.
12     if $k(\textsc{Find}(x)) = \min_{z' \in B_d} k(\textsc{Find}(z'))$ and CYCLE $= 1$ then
13         EXIT LOOP.
14     else
15         set STATUS$(x) = 1$ and Explore-Forward$(x)$.
16     Let $y = \arg\max_{y' \in B_a} k(\textsc{Find}(y'))$.
17     if $k(\textsc{Find}(y)) < \max_{y' \in F_d} k(\textsc{Find}(y'))$ and CYCLE $= 0$ then
18         EXIT LOOP.
19     if $k(\textsc{Find}(y)) < \max_{y' \in F_d} k(\textsc{Find}(y'))$ and CYCLE $= 1$ then
20         EXIT LOOP.
21     if $k(\textsc{Find}(y)) = \max_{y' \in F_d} k(\textsc{Find}(y'))$ and CYCLE $= 1$ then
22         EXIT LOOP.
23     else
24         set STATUS$(y) = 1$ and Explore-Backward$(y)$.
25 if CYCLE $= 0$ then
26     return NO
27 if CYCLE $= 1$ then
28     if the algorithm ended in 12 (or 20) then
29         Let $z^* = \arg\min_{z' \in B_d} k(\textsc{Find}(z'))$ (or $z^* = \arg\max_{z' \in F_d} k(\textsc{Find}(z'))$).
30         Do a DFS backwards from $u$, over the set of vertices $x$ with STATUS$(x) = 1$.
             Mark those that reach some vertex in $C(z^*)$ or $C(v)$.
31         Do a DFS forwards from $v$, over the set of vertices $x$ with STATUS$(x) = 1$.
             Mark those that reach some vertex in $C(v)$ or $C(z^*)$.
32     if If the algorithm ended in 10 (or 18) then
33         Do a forward DFS search from $v$, over the set of vertices $x$ with
             STATUS$(x) = 1$, mark those that reach some vertex in $C(u)$.
34     Let $z$ be a marked canonical vertex.
35     for all canonical $x \neq z$ that is marked do
36         Link$(z, x)$.
```

11 Shiri Chechik, Thomas Dueholm Hansen, Giuseppe F. Italiano, Jakub Lacki, and Nikos Parotsidis. Decremental single-source reachability and strongly connected components in õ(m$\sqrt{n}$) total update time. In Irit Dinur, editor, *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*, pages 315–324. IEEE Computer Society, 2016.

12 Shiri Chechik and Tianyi Zhang. Incremental single source shortest paths in sparse digraphs. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 2463–2477. SIAM, 2021. doi:10.1137/1.9781611976465.146.

13 Edith Cohen, Amos Fiat, Haim Kaplan, and Liam Roditty. A labeling approach to incremental cycle detection. *CoRR*, abs/1310.8381, 2013. arXiv:1310.8381.

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

15 Maximilian Probst Gutenberg, Virginia Vassilevska Williams, and Nicole Wein. New algorithms and hardness for incremental single-source shortest paths in directed graphs. In Konstantin Makarychev, Yury Makarychev, Madhur Tulsiani, Gautam Kamath, and Julia Chuzhoy, editors, *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*, pages 153–166. ACM, 2020.

16 Bernhard Haeupler, Telikepalli Kavitha, Rogers Mathew, Siddhartha Sen, and Robert E. Tarjan. Incremental cycle detection, topological ordering, and strong component maintenance. *ACM Trans. Algorithms*, 8(1), 2012. doi:10.1145/2071379.2071382.

17 Giuseppe F. Italiano. Amortized efficiency of a path retrieval data structure. *Theoretical Computer Science*, 48:273–281, 1986.

18 Irit Katriel and Hans L. Bodlaender. Online topological ordering. *ACM Trans. Algorithms*, 2(3):364–379, 2006. doi:10.1145/1159892.1159896.

19 Jakub Lacki. Improved deterministic algorithms for decremental reachability and strongly connected components. *ACM Trans. Algorithms*, 9(3):27:1–27:15, 2013. doi:10.1145/2483699.2483707.

20 Hsiao-Fei Liu and Kun-Mao Chao. A tight analysis of the katriel-bodlaender algorithm for online topological ordering. *Theor. Comput. Sci.*, 389(1-2):182–189, 2007. doi:10.1016/j.tcs.2007.08.009.

21 Alberto Marchetti-Spaccamela, Umberto Nanni, and Hans Rohnert. Maintaining a topological order under edge insertions. *Inf. Process. Lett.*, 59(1):53–58, 1996. doi:10.1016/0020-0190(96)00075-0.

22 Liam Roditty. Decremental maintenance of strongly connected components. In Sanjeev Khanna, editor, *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 1143–1150. SIAM, 2013.

23 Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)*, 22(2):215–225, 1975.