Incremental SCC Maintenance in Sparse Graphs

Aaron Bernstein*[1], Aditi Dudeja†[1], and Seth Pettie‡[2]

[1]Rutgers University
[2]University of Michigan

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 [BFGT16] and [HKM+12], also generalize to maintain strongly connected components and their topological sort in general directed graphs. The algorithms of [HKM+12] and [BFGT16] have a total update time of $O(m^{3/2})$ and $O(m \cdot \min\left\{m^{1/2}, n^{2/3}\right\})$ respectively, and this is the state of the art for incremental SCC. But the most recent algorithms for incremental cycle detection and topological sort ([BC18] and [BK20]), which yield total (randomized) update time $\tilde{O}(\min\left\{m^{4/3}, n^2\right\})$, 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 [BK20] 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 [BK20] is a structural theorem (first introduced in [BC18]) 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.

*bernstei@gmail.com. Funded by NSF CAREER Grant 1942010.
†aditi.dudeja@rutgers.edu
‡pettie@umich.edu

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 [PK03], deadlock detection [Bel90], circuit evaluation [AHR+90] and scheduling tasks. In many of these applications, the presence of a cycle is not fatal [PK03], 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 [KB06], 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 [LC07] to $O(m^{3/2}+m\sqrt{n}\log n)$. Since then, these problems and incremental SCC have been studied extensively (see for example [AF10],[AFM08], [BFG09],[HKM+12],[BFGT16], [CFKR13],[MNR96]). 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 [HKM+12] 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 [BC18] gave a Las Vegas algorithm with an expected total update time of $O(m\sqrt{n}\log n)$; Bhattacharya and Kulkarni [BK20] combined the balanced search approach of [HKM+12] with the results of [BC18] 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 [BK20] to the case of general directed graphs. Both [BC18] and [BK20] 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 [BC18] 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 [BC18]) is rather unintuitive.

Our contributions are three-fold. We present a new proof of the structural theorem of [BC18], 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 [BK20] can be extended to maintain SCCs in a general graph, 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. [Rod13, Lac13, CHI+16, BPW19]) and a recent algorithm achieves near-optimal $\tilde{O}(m)$ total expected update time [BPW19].

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

Reference	Update Time	Incremental SCC
[KB06]	$O(\min\{m^{3/2}\log n, m^{3/2}+n^2\log n\})$	No
[LC07]	$O(m^{3/2}+m\sqrt{n}\log n)$	No
[AFM08]	$O(n^{2.75})$	No
[BFG09]	$\tilde{O}(n^2)$	No
[HKM+12]	$O(m^{3/2})$	Yes
[BFGT16]	$O(m\cdot\min\{m^{1/2}, n^{2/3}\})$, $\tilde{O}(n^2)$	Yes
[BC18]	$\tilde{O}(m\sqrt{n})$	No
[BK20]	$\tilde{O}(m^{4/3})$	No

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 [GWW20] and $\tilde{O}(m\sqrt{n}+m^{7/5})$ in sparse ones [CZ21].

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 [Ita86] on single-source incremental reachability.

Lemma 2. [Ita86] 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 [BC18] (proved in the appendix of their paper).

Lemma 3. [BC18] 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 [Tar75]. 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, this returns 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.

The 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 [DS87] and [BCD+02], 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 [BC18] introduced the notion of *sometime-τ-similar* pairs. We briefly discuss their definition.

Definition 4. [BC18] 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 [BC18]. However, even for the simpler case of DAGs, the proof in [BC18] 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 $I(v) - I(u) \in [2^i, 2^{i+1})$. □

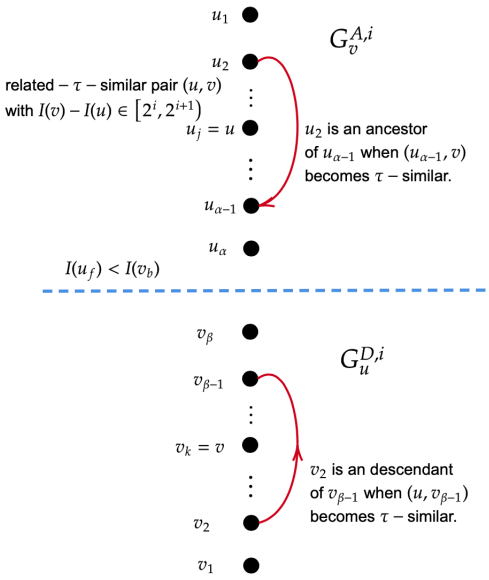

Figure 1: We consider a *related-τ-similar* pair (u, v), where $I(v) - I(u) \in [2^i, 2^{i+1})$. All vertices of $A^i(v)$ appear before the vertices of $D^i(u)$.

Claim 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.

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.

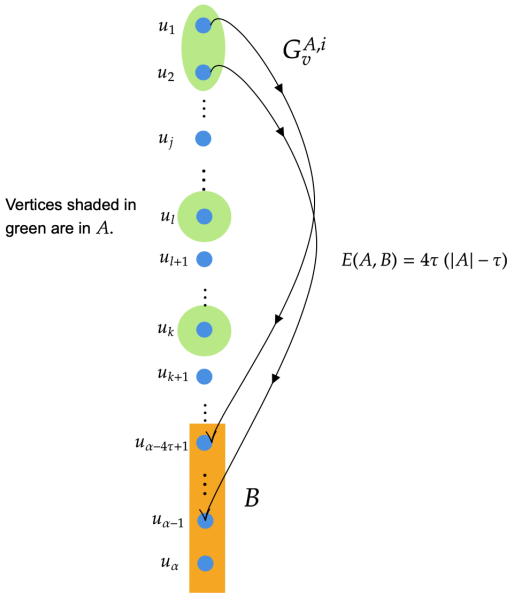

Figure 2: The vertices in green are the vertices of $A^i(v) \setminus B$ that are bad for v. Since the vertices in B are τ-similar to v, the total number of edges coming out of A and going into B is at least $4\tau(|A| - \tau)$.

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 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 [BC18] 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 $|C(y)| \leq \tau$.

[BC18] sample a set S, and maintain 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 $\{i = k \text{ and } j > l\}$ (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 [BC18] 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 [BC18] and [BK20] 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 [BK20]. 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 Appendix A.2 and Appendix A.3 respectively. Phase 1 is essentially the same as in the framework of [BC18], so we postpone the details to Appendix A.1.

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 full description is in the appendix.

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 [BC18], but is modified to handle SCCs.

Lemma 28 ([BC18])**.** Consider the insertion of edge (u, v). Phase 1 has the following guarantees:

(a) 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 $\prec^*$.

(b) 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.

(c) 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(a))

Previous Work. Our Phase 2 will be similar to the cycle detection algorithm of [BK20], but we need to adapt it to find the newly formed strongly connected component. Previous algorithms for finding SCCs such as the one by [HKM+12], 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 [HKM+12].

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(c)). 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.

Algorithm 1. Explore-Forward(x)

1. $F_a = F_a \setminus \{x\}$ and $F_d = F_d \cup \{x\}$.
2. For $x' \in \text{out}(x)$ with $V(x) = V(x')$ do: // see Definition 24.
   (a) If $\textsc{Find}(x') \in B_a \cup B_d$ then $\textsc{cycle} = 1$, and if $x' \notin F_a \cup F_d$, then add x' to F_a.

Algorithm 2. Explore-Backward(x)

1. $B_a = B_a \setminus \{x\}$ and $B_d = B_d \cup \{x\}$.
2. For $x' \in \text{in}(x)$ with $V(x) = V(x')$ do: // see Definition 24.
   (a) If $\textsc{Find}(x') \in F_a \cup F_d$ then $\textsc{cycle} = 1$, and if $x' \notin B_a \cup B_d$, then add x' to B_a.

Algorithm 3. FindComponent(u, v). // Handles insertion of edge (u, v).

1. If $k(\textsc{Find}(u)) < k(\textsc{Find}(v))$, then return no.
2. Initialize $\textsc{cycle} = 0$ and min-heaps $F_a = \{v\}, B_a = \{u\}, F_d = \emptyset, B_d = \emptyset$.
3. If $\textsc{Find}(u) = \textsc{Find}(v)$ or $V(u) \neq V(v)$ then return no.
4. While $F_a \neq \emptyset$ and $B_a \neq \emptyset$ do:
   (a) Let $x = \arg\min_{x' \in F_a} k(\textsc{Find}(x'))$.
   (b) If $k(\textsc{Find}(x)) > \min_{z' \in B_d} k(\textsc{Find}(z'))$ and $\textsc{cycle} = 0$, then exit loop.
   (c) If $k(\textsc{Find}(x)) > \min_{z' \in B_d} k(\textsc{Find}(z'))$ and $\textsc{cycle} = 1$, then exit loop.
   (d) If $k(\textsc{Find}(x)) = \min_{z' \in B_d} k(\textsc{Find}(z'))$ and $\textsc{cycle} = 1$, then exit loop.
   (e) Else set $\textsc{status}(x) = 1$ and Explore-Forward(x).

(f) Let $y = \arg\max_{y' \in B_a} k(\textsc{Find}(y'))$.

(g) If $k(\textsc{Find}(y)) < \max_{y' \in F_d} k(\textsc{Find}(y'))$ and $\textsc{cycle} = 0$, then exit loop.

(h) If $k(\textsc{Find}(y)) < \max_{y' \in F_d} k(\textsc{Find}(y'))$ and $\textsc{cycle} = 1$, then exit loop.

(i) If $k(\textsc{Find}(y)) = \max_{y' \in F_d} k(\textsc{Find}(y'))$ and $\textsc{cycle} = 1$, then exit loop.

(j) Else set $\textsc{status}(y) = 1$ and Explore-Backward(y).

5. If $\textsc{cycle} = 0$, then return no.

6. If $\textsc{cycle} = 1$ then:

   (a) If the algorithm ended in 4d (or 4i) then let $z^* = \arg\min_{z' \in B_d} k(\textsc{Find}(z'))$ (or $z^* = \arg\max_{z' \in F_d} k(\textsc{Find}(z'))$). We then do a search as follows.

      i. Do a DFS backwards from u, over the set of vertices x with $\textsc{status}(x) = 1$. Mark those that reach some vertex in $C(z^*)$ or $C(v)$.

      ii. Do a DFS forwards from v, over the set of vertices x with $\textsc{status}(x) = 1$. Mark those that reach some vertex in $C(v)$ or $C(z^*)$.

   (b) If the algorithm ended in 4c (or 4h), then do a forward DFS search from v, over the set of vertices x with $\textsc{status}(x) = 1$, mark those that reach some vertex in $C(u)$.

   (c) Let z be a marked canonical vertex. For all canonical $x \neq z$ that is marked, $\textsc{Link}(z, x)$.

We postpone the proof of correctness to the appendix. 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 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 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 [BK20]. 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 Property 40 and Property 41 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 UpdateForward and UpdateBackward to update the ordered list. This list only consists of canonical vertices that represent different components.

We now describe how to reorder the vertices. In [BK20] 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 UpdateForward(). The proof for the case when the algorithm terminates in conditions $F_a = \emptyset$ or $\min_{x \in F_a} k(\textsc{Find}(x)) > \min_{y \in B_d} k(\textsc{Find}(y))$ is analogous (we use a subroutine UpdateBackward()) and we omit it here.

Algorithm 4. UpdateForward()

1. $Q = F_d$.
2. $x^* = \arg\max \{k(\textsc{Find}(x)) \mid x \in Q, x \text{ canonical}\}$.
3. $Q = Q \setminus C(x^*)$. // Since we are rearranging canonical vertices.
4. While $Q \neq \emptyset$:
   (a) $x' = \arg\max_{x \in Q} \{k(\textsc{Find}(x))\}$.
   (b) $Q = Q \setminus C(x')$.
   (c) Insert-Before($\textsc{Find}(x'), x^*$)
   (d) $x^* = \textsc{Find}(x')$.
5. $y^* = \textsc{Find}(v)$.
6. $Q = B_d$.
7. While $Q \neq \emptyset$:
   (a) $y' = \arg\max_{y \in B_d} k(\textsc{Find}(y))$.
   (b) $Q = Q \setminus C(y')$.
   (c) Insert-Before($\textsc{Find}(y'), y^*$)
   (d) $y^* = \textsc{Find}(y')$.

We postpone the proof of correctness to the appendix. 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 $\textsc{Find}(x)$ and $\textsc{Find}(y)$ 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 $\max_{x \in B_a} k(\textsc{Find}(x)) \leq \max_{y \in F_d} k(\textsc{Find}(y))$, then we execute UpdateForward(), else if FindComponent() terminated in $F_a = \emptyset$ or $\min_{x \in F_a} k(\textsc{Find}(x)) \geq \min_{y \in B_d} k(\textsc{Find}(y))$, then we execute UpdateBackward(). The proof of correctness can be found in Appendix A.3 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 $\tilde{O}(m^{4/3})$. □

References

[AF10] Deepak Ajwani and Tobias Friedrich. Average-case analysis of incremental topological ordering. *Discrete Appl. Math.*, 158(4):240–250, February 2010.

[AFM08] Deepak Ajwani, Tobias Friedrich, and Ulrich Meyer. An o(n2.75) algorithm for incremental topological ordering. *ACM Trans. Algorithms*, 4(4), August 2008.

[AHR+90] 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.

[BC18] 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.

[BCD+02] 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.

[Bel90] Ferenc Belik. An efficient deadlock avoidance technique. *IEEE Trans. Comput.*, 39(7):882–888, July 1990.

[BFG09] 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.

[BFGT16] 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.

[BK20] 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.

[BPW19] 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.

[CFKR13] Edith Cohen, Amos Fiat, Haim Kaplan, and Liam Roditty. A labeling approach to incremental cycle detection. *CoRR*, abs/1310.8381, 2013.

[CHI+16] Shiri Chechik, Thomas Dueholm Hansen, Giuseppe F. Italiano, Jakub Lacki, and Nikos Parotsidis. Decremental single-source reachability and strongly connected components in $\tilde{o}(m\sqrt{n})$ 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.

[CZ21] 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.

[DS87] 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.

[GWW20] 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.

[HKM+12] Bernhard Haeupler, Telikepalli Kavitha, Rogers Mathew, Siddhartha Sen, and Robert E. Tarjan. Incremental cycle detection, topological ordering, and strong component maintenance. 8(1), 2012.

[Ita86] Giuseppe F. Italiano. Amortized efficiency of a path retrieval data structure. *Theoretical Computer Science*, 48:273–281, 1986.

[KB06] Irit Katriel and Hans L. Bodlaender. Online topological ordering. *ACM Trans. Algorithms*, 2(3):364–379, 2006.

[Lac13] Jakub Lacki. Improved deterministic algorithms for decremental reachability and strongly connected components. *ACM Trans. Algorithms*, 9(3):27:1–27:15, 2013.

[LC07] 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.

[MNR96] Alberto Marchetti-Spaccamela, Umberto Nanni, and Hans Rohnert. Maintaining a topological order under edge insertions. *Inf. Process. Lett.*, 59(1):53–58, 1996.

[PK03] David J. Pearce and Paul H. J. Kelly. Online algorithms for topological order and strongly connected components. Technical report, 2003.

[Rod13] 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.

[Tar75] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)*, 22(2):215–225, 1975.

A Omitted Proofs

A.1 Phase 1 of the Algorithm

For the sake of completeness, we restate the framework of [BC18] which allows us to update $V_{i,j}$ so that they are consistent with $\prec^*$. After addition of the edge (u, v), the algorithm first updates for each $w \in V$, the set $V(w)$. To do this, the algorithm will use Lemma 2. Using this lemma, the algorithm maintains $A(s)$ and $D(s)$ for all $s \in S$ in time $O(m|S|)$, where $|S| = \tilde{O}(n/\tau)$ with high probability. So, when a vertex w is added to $A(s)$ for some $s \in S$, the algorithm adds s to $A_S(w)$. The set $D_S(w)$ is similarly updated. For each vertex w, we maintain counters for $|A_S(w)|$ and $|D_S(w)|$. Therefore, we know what the new partition $V_{i,j}$ for w is.

After updating the partitions of the vertices, we also need to sort the canonical vertices within each $V_{i,j}$. This done by moving the canonical vertices in the ordered list. To implement the algorithm, [BC18] maintain dummy nodes $f_{i,j}$ and $b_{i,j}$ for every partition $V_{i,j}$. It is ensured that for all canonical $x \in V_{i,j}$, $k(f_{i,j}) < k(x) < k(b_{i,j})$. The dummy nodes themselves are ordered in the following way: if $V_{i,j} \prec^* V_{k,l}$ then, $k(f_{i,j}) < k(b_{i,j}) < k(f_{k,l}) < k(b_{k,l})$. We now describe their approach. Let G_{t-1} denote the graph just before the edge (u, v) is inserted, and let G_t denote that graph right after the edge (u, v) is inserted. For a canonical vertex x, let $V^{t-1}(x)$ denote $V(x)$ before the insertion of (u, v) and let $V^t(x)$ denote $V(x)$ after the insertion of edge (u, v). Additionally, they define sets $\text{UP}(i, j) = \{x \mid x \text{ is canonical and } V^{t-1}(x) \prec^* V^t(x) = V_{i,j}\}$ and $\text{DOWN}(i, j) = \{x \mid x \text{ is canonical and } V_{i,j} = V^t(x) \prec^* V^{t-1}(x)\}$. We implement these sets as max-heaps and min-heaps respectively. Once these sets are determined we execute the following algorithms for each $\text{UP}(i, j)$ and $\text{DOWN}(i, j)$.

Algorithm 5. MOVE-UP(i, j)

1. while $\text{UP}(i, j) \neq \emptyset$ do:
   (a) $x = \arg\max_{x' \in \text{UP}(i,j)} k(x')$.
   (b) INSERT-AFTER$(x, f_{i,j})$.
   (c) $\text{UP}(i, j) = \text{UP}(i, j) \setminus \{x\}$.

Algorithm 6. MOVE-DOWN(i, j)

1. while $\text{DOWN}(i, j) \neq \emptyset$ do:
   (a) $x = \arg\min_{x' \in \text{DOWN}(i,j)} k(x')$.
   (b) INSERT-BEFORE$(x, b_{i,j})$.
   (c) $\text{DOWN}(i, j) = \text{DOWN}(i, j) \setminus \{x\}$.

We show that the ordered list output by the algorithm is a topological ordering of the strongly connected components of G^{t-1}.

Lemma 33. Consider any edge (x, y) in G^{t-1} such that at time t, $C(x) \neq C(y)$, then $k(\textsc{Find}(x)) < k(\textsc{Find}(y))$ in the list output by the algorithm.

Proof. We consider two cases: $V^t(\textsc{Find}(x)) \neq V^t(\textsc{Find}(y))$. In this case, $V^t(\textsc{Find}(x)) \prec^* V^t(\textsc{Find}(y))$, and while updating $\{V_{i,j}\}$, $\textsc{Find}(x)$ and $\textsc{Find}(y)$ are placed between the right placeholders, so $k(\textsc{Find}(x)) < k(\textsc{Find}(y))$. Suppose $V^t(x) = V^t(y) = V_{i,j}$. Since $(x, y) \in G^{t-1}$, $k(\textsc{Find}(x)) < k(\textsc{Find}(y))$ in at time $t-1$. We consider the following three cases.

1. $\textsc{Find}(x), \textsc{Find}(y) \in \textsc{Up}(i, j)$. In this case, $\textsc{Move-Up}(i, j)$ puts $\textsc{Find}(x)$, before $\textsc{Find}(y)$, since $\textsc{Move-Up}()$ doesn't change the relative ordering of the canonical vertices in $\textsc{Up}(i, j)$.

2. $\textsc{Find}(x), \textsc{Find}(y) \in \textsc{DOWN}(i, j)$. In this case, $\textsc{Move-Down}(i, j)$ puts $\textsc{Find}(x)$ before $\textsc{Find}(y)$, since $\textsc{Move-Down}()$ doesn't change the relative ordering of the canonical vertices in $\textsc{Down}(i, j)$.

3. $\textsc{Find}(x) \in \textsc{Up}(i, j)$ and $\textsc{Find}(y) \in \textsc{Down}(i, j)$. In this case, the algorithm puts $\textsc{Find}(x)$ after $f_{i,j}$ and $\textsc{Find}(y)$ before $b_{i,j}$. So, $k(\textsc{Find}(x)) < k(\textsc{Find}(y))$.

□

As discussed before, in Phase 1, while updating $A(s)$ and $D(s)$ for all $s \in S$, we are also able to detect if the new strongly connected component (if any) contains a vertex of S. If the newly formed strongly connected component has this property, then we are able to update all its vertices correctly. We briefly discuss how this is done. Let t be the time at which the edge (u, v) is added to the graph.

1. Consider a vertex $x \in D(s)$ before time t that was added to $A(s)$ for some $s \in S$ while updating $A(s)$ at time t, then mark $\textsc{Find}(x)$ and $\textsc{Find}(s)$. Similarly, if a vertex $x \in A(s)$ before time t was added to $D(s)$ at time t for some $s \in S$ while updating $D(s)$, then mark $\textsc{Find}(x)$ and $\textsc{Find}(s)$.

2. Let z be an arbitrary marked canonical vertex. For all canonical vertices $x \neq z$ that are marked, $\textsc{Link}(z, x)$. Delete all $x \neq z$ that are marked from the ordered list.

Note that for a vertex $s \in S$, $\textsc{Find}(s)$ is marked if and only if there is a newly formed strongly connected component, C_N containing $C(s)$. Let $C_1, \cdots, C_r$ be the components that combine to form C_N, and let $x_1, \cdots, x_r$ be the corresponding canonical vertices. Without loss of generality, assume C_1 is the component containing a vertex from S, and let this vertex be s. Observe that $x_2, \cdots, x_r$ are related to s before the addition of edge (u, v). So, $x_i \in A(s)$ (or $D(s)$) before the addition of edge (u, v), and after the addition of edge (u, v), $x_i \in D(s)$ (or $A(s)$) as well. This implies that all x_i's will be marked, and will be linked. Moreover, note that no canonical vertex $y \notin \{x_1, \cdots, x_r\}$ is marked since at least one of these is true at time t: $y \notin A(s)$ or $y \notin D(s)$. From this diswe conclude that C_N is updated correctly after the insertion of (u, v).

Finally, we observe that the newly strongly connected component C_N contains a vertex $s \in S$, then the reordering procedure done by $\textsc{Move-Up}()$ and $\textsc{Move-Down}()$ gives a topological sort of the canonical vertices of the current graph. Towards proving this we consider the following cases.

1. Consider edge (x, y) such that $C(x) \neq C(y)$ and $C(x) \neq C_N$, $C(y) \neq C_N$ at time t. In this case, we know from Lemma 33 that $k(\textsc{Find}(x)) < k(\textsc{Find}(y))$.

2. Consider edge (x, y) such that $C(x) \neq C(y)$, and $C(x) = C_N$ at time t. In this case, observe that $A_S(x) \subset A_S(y)$. Additionally, we know that $s \in D_S(x)$ but $s \notin D_S(y)$. This implies that $V^t(\textsc{Find}(x)) \prec^* V^t(\textsc{Find}(y))$, and therefore, $k(\textsc{Find})(x) < k(\textsc{Find}(y))$.

3. Consider edge (x, y) such that $C(x) \neq C(y)$ and $C(y) = C_N$ at time t. This is analogous to the case above.

In the above three cases, we have accounted for all types of edges, and this proves that we have a topological sort of the canonical vertices.

Lemma 34. The total runtime of Phase 1 over m edge insertions is $O(\frac{m \cdot n}{\tau})$.

Proof. For each vertex x, the number of times $V(x)$ can change is $O(n/\tau)$. For each vertex that changes its bucket, moving it to the correct bucket using MOVE-UP() and MOVE-DOWN() routine takes time $O(1)$ per vertex. This leads to a total runtime of $O(n^2/\tau)$ over all edge insertions. Additionally, we maintain $A(s)$ and $D(s)$ for all $s \in S$, this takes time $O(m \cdot n/\tau)$ over all insertions. We can mark all vertices of C_N and link them in time $O(m \cdot n/\tau)$ time over insertion of all edges. This proves our claim. □

A.2 Phase 2 of the Algorithm

In this case, we prove correctness of Phase 2. We first start with proving some basic properties of FINDCOMPONENT().

Basic Properties of the Algorithm. We will now show that Phase 2 outputs NO if and only if a new component is not formed. In addition, it also links all the canonical vertices of the newly formed component correctly. Before we do this, we state some properties of the algorithm.

Property 35. Sets F_d and F_a are mutually exclusive, and so are sets B_d and B_a.

Proof. A vertex is added to F_d only after it is removed from F_a. Similarly, a vertex is added to B_d after it is removed from B_a. □

Property 36. The variable CYCLE $= 1$ if and only if there exists $x \in B_a \cup B_d$ and $y \in F_a \cup F_d$ such that FIND(x) = FIND(y).

Proof. We set the variable CYCLE in EXPLORE-FORWARD and EXPLORE-BACKWARD. In the former case, it is done when we are forward exploring a vertex x' and $x' \in F_a \cup F_d$ or is added to it F_a, and FIND$(x') \in B_d \cup B_a$ already. In the latter case it is done when we are backward exploring a vertex x' and $x' \in B_a \cup B_d$ or is added to B_a, and FIND$(x') \in F_d \cup F_a$ already.

To prove the other direction, suppose without loss of generality that there exists $x' \in F_a \cup F_d$ and $y' \in B_a \cup B_d = B$ such that FIND(x') = FIND(y'). Suppose x' is added to F_a after it was added to $B_a \cup B_d$, then, while adding x' to F_a, the algorithm will check condition 2a of EXPLORE-FORWARD and update CYCLE. The case where x' is added to B_a after it is added to F_a is also analogous. □

Property 37. At all times $\min_{y \in B_d} k(\text{FIND}(y)) > \max_{z \in F_d} k(\text{FIND}(z))$.

Proof. This is true when the sets are empty at time $t = 0$. Suppose the claim is true till time $t - 1$ and first becomes false at time t. There are two possibilities.

1. A vertex x is explored forward and is added to F_d. By inductive hypothesis, the claim held before time t. This implies that at time t, $k(\text{FIND}(x)) \geq \min_{y \in B_d} k(\text{FIND}(y))$. We show that the algorithm would not have added x to F_d. To see this consider two cases: If $k(\text{FIND}(x)) > \min_{y \in B_d} k(\text{FIND}(y))$, then when x is explored, either condition 4c or 4b is encountered, and we exit the loop instead of adding x to F_d. On the other hand, another possibility is that $k(\text{FIND}(x)) = \min_{y \in B_d} k(\text{FIND}(y))$. In this case, we know that CYCLE $= 1$ from Property 36. In this case, we encounter condition 4d, which asks us to exit the loop instead of adding x to F_d.

2. A vertex y is explored backward and added to B_d. The proof is analogous to the case above.

This completes our proof. □

From Property 37, we have the following property.

Property 38. Let $x \in F_d$ and $y \in B_d$, then FIND$(x) \neq$ FIND(y).

Property 39. If CYCLE $= 0$, then all four sets F_a, B_a, F_d, B_d are pairwise mutually exclusive.

Proof. From Property 38, we know that F_d and B_d are pairwise mutually exclusive. From Property 35, we conclude that F_a and F_d are pairwise mutually exclusive, and that B_a and B_d are pairwise mutually exclusive. From Property 36, we conclude that if CYCLE $= 0$, then, F_a is disjoint from $B_a \cup B_d$ and B_a is disjoint from $F_a \cup F_d$. This concludes our proof. □

Property 40. Consider two vertices x, y such that $V(x) = V(y)$. Suppose there is a path from v to x, and a path from v to y. If $k(\textsc{Find}(x)) < k(\textsc{Find}(y))$, and $y \in F_d$, then $x \in F_d$ as well.

Property 41. Consider two vertices x, y such that $V(x) = V(y)$. Suppose there is a path from x to u, and a path from y to u. If $k(\textsc{Find}(x)) < k(\textsc{Find}(y))$, and $x \in B_d$, then $y \in B_d$ as well.

A.2.1 Correctness of the Algorithm.

We now show correctness of our algorithm. To do this, we will first show that the algorithm correctly determines if a new strongly connected component has been formed. Then, we show that if we are in the case where a new component has been formed, then the algorithm correctly updates this component.

Lemma 42. The algorithm outputs NO if and only if there is no newly formed strongly connected component in the graph.

Proof. To prove this statement, we analyze different stopping conditions:

1. CYCLE $= 1$. As discussed in Property 36, we note that there exists $x \in F$ and $y \in B$ such that $\textsc{Find}(x) = \textsc{Find}(y)$. This implies that the component containing x and y, has a path to u, and is also reachable from v. So, there is a newly formed strongly connected component, and in this case, the algorithm doesn't output NO.

2. CYCLE $= 0$. We show that there is no new strongly connected component formed. We analyze different cases.

   (a) $F_a = \emptyset$. Suppose the graph contains a newly formed strongly connected component. This implies that there is a v to u path, denoted P_{vu}. Note that $v \in F_a \cup F_d = F_d$, $u \in B_a \cup B_d$. Since CYCLE $= 0$, from Property 39, we conclude that $u \notin F_d$. So, there is a vertex on P_{vu} that doesn't belong in F_d. Let x be the first vertex on this path such that $x \notin F_d$. Consider the predecessor $p(x)$ of x on this path. Note that $p(x) \in F_d$. This implies that $x \in F_d \cup F_a = F_a$. This is a contradiction to the fact that $F_a = \emptyset$.

   (b) $B_a = \emptyset$. The argument in this case is analogous to the case above.

   (c) $\min_{x \in F_a} k(\textsc{Find}(x)) > \min_{y \in B_d} k(\textsc{Find}(y))$. Suppose there is a newly formed strongly connected component. This implies there is a path P_{vu} from v to u. Suppose there is a vertex x^* on P_{vu} that lies in F_a. Note that $k(\textsc{Find}(x^*)) > \min_{y \in B_d} k(\textsc{Find}(y))$. This implies that $x^* \in B_d$ from Property 41. This contradicts $F_a \cap B_d = \emptyset$ implied by Property 39.
   We now are left to show that there is a vertex x^* on the path P_{vu} that lies in F_a. If $v \in F_a$, then we are done. Otherwise, we know that $v \in F_d$. As before, note that $u \in B_d \cup B_a$, and $u \notin F_d$. This is because of Property 39. Let x' be the first vertex on P_{vu} that doesn't lie in F_d. Consider the predecessor $p(x')$ of x' on P_{vu}, $p(x') \in F_d$ ($p(x')$ lies on P_{vu} because $x' \neq v$). This implies $x' \in F_a \cup F_d = F_a$.

   (d) $\max_{x \in B_a} k(\textsc{Find}(x)) < \max_{y \in F_d} k(\textsc{Find}(y))$. This argument is analogous to the case above.

□

We now prove some claims towards showing that the strongly connected components are updated correctly.

Claim 43. Suppose x is a canonical vertex that is linked by FINDCOMPONENT(), then $x \in C_N$, where C_N is the new strongly connected component.

Proof. Note that z^*, is reachable from v, and has a path to u as well. DFS marks and links only vertices that lie on some $v - u$ path, or a $v - z^*$ path, or a $z^* - u$ path. All these vertices are contained in the newly formed strongly connected component. □

Lemma 44. Suppose the algorithm ends the loop due to the condition $\min_{x \in F_a} k(\textsc{Find}(x)) = \min_{y \in B_d} k(\textsc{Find}(y))$, then for all $z \in V$, at least one of these three statements if true.

1. $\textsc{Find}(z^*) = \textsc{Find}(z)$, where $z^* = \arg\min_{x \in B_d} k(\textsc{Find}(x))$.

2. The newly formed strongly connected component does not contain z.

3. $z \in F_d \cup B_d$.

Proof. Suppose there is z such that $\textsc{Find}(z) \neq \textsc{Find}(z^*)$, but z lies in the newly created strongly connected component. Note that $k(\textsc{Find}(z)) \neq k(\textsc{Find}(z^*))$, we want to show that $z \in F_d \cup B_d$. This brings us to the following two cases:

1. $k(\textsc{Find}(z)) > k(\textsc{Find}(z^*))$. In this case, since z is in the newly created strongly connected component, there is a path from z to u. So, application of Property 41, $z \in B_d$.

2. $k(\textsc{Find}(z)) < k(\textsc{Find}(z^*))$. Suppose $z \notin F_d$. Consider a $v - z$ path P_{vz} (such a path exists since z belongs in the newly created strongly connected component). We claim that there is a vertex on P_{vz} that lies in F_a. If $v \in F_a$, then we are done. Else, $v \in F_d$. We conclude that there is a vertex on P_{vz} that lies in F_d. Let a be such a vertex that is farthest from v. Note that $a \neq z$, so there is a successor of a on P_{vz}. Let this successor be $s(a)$. Note that $s(a) \in F_d \cup F_a = F_a$. This implies that $k(\textsc{Find}(s(a))) \leq k(\textsc{Find}(z)) < k(\textsc{Find}(z^*))$. Since $s(a) \in F_a$, we know that $\min_{x \in F_a} k(\textsc{Find}(x)) < k(\textsc{Find}(z^*)) = \min_{y \in B_d} k(\textsc{Find}(y))$. This contradicts the statement of the lemma.

This implies that $z \in B_d \cup F_d$, which proves our claim. □

We have the following mirror lemma.

Lemma 45. Suppose the algorithm ends the loop due to the condition $\max_{x \in B_a} k(\textsc{Find}(x)) = \max_{y \in F_d} k(\textsc{Find}(y))$, then for all $z \in V$, at least one of these three statements if true.

1. $\textsc{Find}(z^*) = \textsc{Find}(z)$, where $z^* = \arg\max_{x \in F_d} k(\textsc{Find}(x))$.

2. The newly formed strongly connected component does not contain z.

3. $z \in F_d \cup B_d$.

To show that all vertices that belong in the newly created strongly connected component C_N are linked, it is sufficient to show that for all vertices $x \in C_N$, either $\textsc{status}(x) = 1$ or $x \in C(z^*)$.

Corollary 46. Let C_N be the newly formed strongly connected component. From Lemma 44 and Lemma 45, we conclude that if the algorithm concludes in 4i or 4d, then for all $x \in C_N$, either $x \in F_d \cup B_d$, which means that $\textsc{status}(x) = 1$, or $x \in C(z^*)$. This implies that the algorithm will link $\textsc{Find}(x)$ in the connected component it outputs.

Now, we are concerned with the case when the algorithm doesn't terminate in conditions 4i or 4d. Towards this, we prove the following lemma:

Lemma 47. Let C_N be the newly created strongly connected component and suppose the algorithm doesn't terminate in 4i or 4d, then for all $x \in C_N$, $\textsc{status}(x) = 1$.

Proof. Suppose there is a vertex z such that $z \in C_N$, but $\textsc{status}(z) = 0$, and the algorithm doesn't terminate in 4i or 4d. We will argue that if this is true, then the algorithm doesn't terminate. We first show that $F_a \neq \emptyset$ and $B_a \neq \emptyset$. Consider the path P_{vzu}. If $v \in F_a$ then we are done. So, we assume that $v \in F_d$. Note that P_{vz} contains a vertex that is not in F_d (since z is such a vertex, because $\textsc{status}(z) = 0$). Let f be the first such vertex on this path. Consider the predecessor of f, called $p(f)$. We know that $p(f) \in F_d$. So, $f \in F_d \cup F_a = F_a$. Similarly, we can show that there exists a vertex on the path P_{zu}, called b such that $b \in B_a$. We further observe that $k(\textsc{Find}(f)) \leq k(\textsc{Find}(z)) \leq k(\textsc{Find}(b))$. We now show that the algorithm cannot terminate in other conditions as well. Towards this, we examine two possibilities.

1. $k(\textsc{Find}(f)) < k(\textsc{Find}(b))$. In this case, note that we have the following set of inequalities.

   (a) $\min_{x \in F_a} k(\textsc{Find}(x)) \leq k(\textsc{Find}(f)) < k(\textsc{Find}(b)) \leq \min_{y \in B_d} k(\textsc{Find}(y))$. The first inequality is due to the fact that $f \in F_a$, and the final inequality is due to the fact that if $k(\textsc{Find}(b)) > \min_{y \in B_d} k(\textsc{Find}(y))$, then $b \in B_d$ ($b \in C_N$ and using Property 41). So, the algorithm doesn't terminate in condition 4c or 4b.

   (b) $\max_{x \in F_d} k(\textsc{Find}(x)) \leq k(\textsc{Find}(f)) < k(\textsc{Find}(b)) \leq \max_{y \in B_a} k(\textsc{Find}(y))$. The first inequality is due to the fact that if $k(\textsc{Find}(f)) < \max_{x \in F_d} k(\textsc{Find}(x))$, then $f \in F_d$ as well ($f \in C_N$ and using Property 40). The second inequality is due to the fact that $b \in B_a$. So, the algorithm doesn't terminate in condition 4h or 4g.

2. $k(\textsc{Find}(f)) = k(\textsc{Find}(b))$. We again consider two set of inequalities.

   (a) $\min_{x \in F_a} k(\textsc{Find}(x)) \leq k(\textsc{Find}(f)) = k(\textsc{Find}(b)) \leq \min_{y \in B_d} k(\textsc{Find}(y))$.

   (b) $\max_{x \in F_d} k(\textsc{Find}(x)) \leq k(\textsc{Find}(f)) = k(\textsc{Find}(b)) \leq \max_{y \in B_a} k(\textsc{Find}(y))$.

   Since we don't terminate in 4d or 4i, one of the inequalities in each of 2a and 2b are strict. On the other hand, if these inequalities are strict, then we don't terminate in 4c or 4b or 4h or 4g. This is a contradiction.

□

Lemma 48. Let C_N be the new strongly connected component formed due the the addition of edge (u, v). Then, a canonical vertex x is linked by FindComponent() if and only if x is present in C_N.

Proof. From Claim 43 we conclude that all canonical vertices that are linked are indeed present in the newly formed strongly connected component. From Corollary 46 and Lemma 47 we conclude that all the vertices in the newly formed strongly connected components are linked. □

A.2.2 Runtime of Phase 2

The following lemma is implied by the fact that we alternate between forward and backward searches.

Lemma 49. Let edge e_t be inserted at time t, and consider sets F_d and B_d after e_t has been processed, then $|F_d| - 1 \leq |B_d| \leq |F_d| + 1$.

Definition 50. Let e_l be the edge inserted at time l, and suppose f_l denotes the number of vertices in F_d, after e_l is processed.

Note that the the total runtime of the algorithm is $O(m/n \sum_{l=0}^{m} f_l)$ (due to Lemma 3). To bound this sum, we first show the following lemma:

Lemma 51. $\sum_{l=0}^{m} f_l^2 = \tilde{O}(n\tau)$.

Proof. Consider F_d and B_d after we have we have processed the edge e_t. By Lemma 49, we know that both these sets have size $\Theta(f_t)$. To prove the lemma, we show that for every $x \in F_d$ and $y \in B_d$, the ordered pair (x, y) is either a new *related-sometime-τ-similar* pair or *equivalent-sometime-τ-similar* pair. We consider the following two cases.

1. Let t be a time during which a strongly connected component is not formed. Suppose (u, v) is the edge that is processed at time t, and let G be the graph after the addition of edge (u, v). Note that after Phase 1, the ordered list of strongly connected components corresponds to a topological ordering of the strongly connected components of $G \setminus \{(u, v)\}$ (see Lemma 28). Consider $x \in F_d$ and $y \in B_d$, from Property 37, we know that $k(\textsc{Find}(x)) < k(\textsc{Find}(y))$. Note that due to Lemma 28, there is no path from y to x in $G \setminus \{(u, v)\}$. Further, there can't be a path from x to y, since a new strongly connected component containing them isn't created by the addition of (u, v). This implies that x and y were not related before time t. On the other hand, at time t, they are S-equivalent of *Type 2*, since x and y are related, $V(x) = V(y)$ and $|C(x)| \leq \tau, |C(y)| \leq \tau$ (from Observation 29 and Lemma 25). This implies that with high probability, they are *related-sometime-τ-similar* (from Lemma 26, and using the fact that a new component is not formed). So, the number of newly formed *sometime-τ-similar* pairs at time t is f_t^2.

2. Next, we consider times t when a new strongly connected component is formed. We recall that since we are in Phase 2, the size of the newly formed strongly connected component C_N is at most τ. To analyze this scenario, we can therefore, condition on Lemma 26. We consider the following cases.

   (a) **When $x \in F_d, y \in B_d$ and $x, y \notin C_N$.** From Property 37 we conclude that $k(\textsc{Find}(x)) < k(\textsc{Find}(y))$ before the insertion of (u, v). Consequently from Lemma 28, we conclude there couldn't have been a path from y to x in $G \setminus \{(u, v)\}$. Moreover, if there was a path from x to y, then $x, y \in C_N$, since the path from v to x, x to y, y to u, combined with (u, v) would have given us a cycle. So, in this case, x and y were not related before the insertion of (u, v). After the insertion of (u, v), x and y have become a *related-τ-similar* pair.

(b) **When** $x \in F_d, y \in B_d$ **and** $x \in C_N$, $y \notin C_N$. As before, $k(\textsc{Find}(x)) < k(\textsc{Find}(y))$, which tells us that there is no path from y to x in $G \setminus \{(u,v)\}$ (because of Lemma 28). Similar to the argument above, if there was a path from x to y in $G \setminus \{(u,v)\}$, then this path combined with the path from v to x, y to u and the edge (u,v) would have implied that $x \in C_N$ as well. So, x and y were not related before the insertion of (u,v) and were therefore, not a τ-similar pair. They become a *related-τ-similar* pair after the insertion of (u,v).

(c) **When** $x \in F_d, y \in B_d$ **and** $x \notin C_N$, $y \in C_N$. Similar to the above case, we can argue that x and y were not τ-similar before the insertion of (u,v) but are *related-τ-similar* now.

(d) **When** $x \in F_d, y \in B_d$ **and** $x \in C_N$, $y \in C_N$. From Property 38, we can conclude that x and y were not *equivalent-sometime-τ-similar* before the insertion of (u,v), but are *equivalent-sometime-τ-similar* now since $|C_N| \leq \tau$.

From the above cases, we can conclude that after processing an edge (u,v), if we consider $x \in F_d$, $y \in B_d$, then we can charge them to a *equivalent-sometime-τ-similar* pair or *related-sometime-τ-similar* pair (x,y), which wasn't charged before. Since the total number of *sometime-τ-similar* pairs are at most $\tilde{O}(n\tau)$, we conclude that $\sum_{t=0}^{m} f_t^2 = \tilde{O}(n\tau)$. □

We now show an upper bound on the runtime of Phase 2.

Lemma 52. $\sum_{l=0}^{m} f_l = O(\sqrt{mn\tau})$. The total runtime of phase 2 is $O(\sqrt{m^3\tau/n})$.

Proof. We first note from Lemma 3, that the degree of every vertex is $O(m/n)$. The runtime of the algorithm is $O(m/n \sum_{l=0}^{m} f_l)$. Applying Cauchy-Schwarz to Lemma 51, we note that $\sum_{l=0}^{m} f_l = O(\sqrt{mn\tau})$. This gives us the required bound on the runtime of Phase 2. □

A.3 Phase 3 of the Algorithm

We show that Update-Forward() correctly orders the canonical vertices in a topological order.

Correctness. Let $S_f = \{x \text{ canonical} \mid k(x) < k(x_f)\} \setminus F_d$ and $S_b = \{y \text{ canonical} \mid k(y) > k(x_f)\} \setminus B_d$. We first have the following observation:

Observation 53. Consider any canonical $x, y \in S_f \cup S_b$. If $k(x) < k(y)$ before the reordering of canonical vertices, then $k(x) < k(y)$ after the reordering.

Observation 54. Consider any canonical $x, y \in F_d$ (or canonical $x, y \in B_d$). If $k(x) < k(y)$ before reordering, then $k(x) < k(y)$ after the reordering as well since Update-Forward() does not change the relative order of the canonical vertices in F_d. Similarly, it doesn't change the relative order of canonical vertices in B_d.

Observation 55. There is no edge $(x,y) \in G \setminus \{(u,v)\}$ such that $x \in B_d$ and $y \in F_d$ (see Property 37 and Lemma 28). There is no edge $(x,y) \in G \setminus \{u,v\}$ such that $x \in F_d$, $y \in B_d$ since we are dealing with the case when no new components are formed. Therefore, the only edge between F_d and B_d is (u,v), and $k(\textsc{Find}(u)) < k(\textsc{Find}(v))$ after reordering.

Lemma 56. Consider any edge (x,y) in G between sets $F_d \cup B_d$ and $S_b \cup S_f$, then after the run of Update-Forward(), $k(\textsc{Find}(x)) < k(\textsc{Find}(y))$.

Proof. We consider the following cases. Throughout the proof, it will be useful to refer to Figure 3.

1. **Edges between F_d and S_f.** Consider (x,y) such that $\textsc{Find}(x) \in S_f$ and $y \in F_d$, then $k(\textsc{Find}(x)) < k(\textsc{Find}(y))$ after reordering. We now consider edges (x,y) such that $x \in F_d$ and $\textsc{Find}(y) \in S_f$. We examine two cases.

   (a) **When** $V(x) \neq V(y)$. In this case from Lemma 28, we conclude that $k(x_f) < k(y)$, thus contradicting the fact that $\textsc{Find}(y) \in S_f$.

   (b) **When** $V(x) = V(y)$. In this case, we know that $y \in F_a$, and therefore, $k(x_f) < k(\textsc{Find}(y))$, which contradicts the fact that $\textsc{Find}(y) \in S_f$.

2. **Edges between F_d and S_b.** Consider $x \in F_d$ and $\textsc{Find}(y) \in S_b$, $k(\textsc{Find}(x)) < k(\textsc{Find}(y))$ both before and after the reordering procedure.

3. **Edges between S_f and B_d.** Consider $\textsc{Find}(x) \in S_f$ and $y \in B_d$, $k(\textsc{Find}(x)) < k(\textsc{Find}(y))$ both before and after the reordering procedure.

4. **Edges between S_b and B_d.** Consider edge (x, y) with $x \in B_d$ and $\textsc{Find}(y) \in S_b$, $k(\textsc{Find}(x)) < k(\textsc{Find}(y))$ after reordering the vertices. We now show that there are no edges (x, y) such that $\textsc{Find}(x) \in S_b$ and $y \in B_d$. We consider two cases.

   (a) **When $V(x) = V(y)$.** In this case, $x \in B_a$ (since when we added y to B_d, we made sure all in-neighbours x' of y such that $V(x') = V(y)$ were present in $B_a \cup B_d$) and $k(x_f) < k(\textsc{Find}(x)) < k(y_1)$. Since $B_a \neq \emptyset$, the algorithm terminates in the condition $\max_{z \in B_a} k(\textsc{Find}(z)) < \max_{w \in F_d} k(\textsc{Find}(w)) = k(x_f)$. On the other hand, $k(\textsc{Find}(x)) \leq \max_{z \in B_a} k(\textsc{Find}(z))$. This implies that $k(\textsc{Find}(x)) < k(x_f)$, which is a contradiction.

   (b) **When $V(x) \neq V(y)$.** Since $V(y) = V(x_f)$, we conclude that $V(x_f) \neq V(x)$ as well. Additionally, since (x, y) is an edge, this implies that $V(x) \prec^* V(x_f)$, and by Lemma 28 we conclude that $k(\textsc{Find}(x)) < k(x_f)$, thus contradicting the fact that $\textsc{Find}(x) \in S_b$.

□

Finally, we consider the case when there is a newly formed strongly connected component. In this case, we delete all the non-canonical vertices of the newly formed strongly connected component in the ordered list and run UpdateForward when the algorithm terminates in conditions $B_a = \emptyset$ or $\max_{x \in B_a} k(\textsc{Find}(x)) \leq \max_{y \in F_d} k(\textsc{Find}(y))$. Without loss of generality, assume that v is the canonical vertex of the new strongly connected component. Let $v, x_1, x_2, \cdots, x_f$ be the canonical vertices present in F_d after processing (u, v) and deleting the non-canonical vertices of the newly formed component, such that $k(v) < k(x_1) < \cdots < k(x_f)$. Similarly, let $y_1, y_2, \cdots, y_b$ be the canonical vertices in B_d with $k(y_1) < k(y_2) < \cdots < k(y_b)$. By Property 37, we know that $k(x_f) < k(y_1)$. We now briefly describe why the same algorithm is correct for this case as well. We define sets S_f and S_b as before: $S_f = \{x \text{ canonical} \mid k(x) < k(x_f)\} \setminus F_d$ and $S_b = \{y \text{ canonical} \mid k(y) > k(x_f)\} \setminus B_d$.

1. **Edges between F_d and B_d.** Note that there are no edges of the type (w, z) such that $\textsc{Find}(w) = x_i$ for some i and $\textsc{Find}(z) = y_j$ for some j The presence of such an edge would imply that these vertices would be a part of the newly formed component. Addition of the edge (u, v), implies that $y_b < v$ in the topological ordering of the new graph. So, the new ordering of the canonical vertices, $y_1, y_2, \cdots, y_b, v, x_1, \cdots, x_f$ is valid for edges between these two sets.

2. **Edges between vertices of $S_b \cup S_f$.** Consider any x, y such that $\textsc{Find}(x), \textsc{Find}(y) \in S_f \cup S_b$. If $k(\textsc{Find}(x)) < k(\textsc{Find}(y))$ before reordering the canonical vertices, then the same is true after reordering.

3. Consider any $x, y \in F_d$ (or $x, y \in B_d$). If $k(\textsc{Find}(x)) < k(\textsc{Find}(y))$ before reordering, then $k(\textsc{Find}(x)) < k(\textsc{Find}(y))$ after the reordering as well.

4. **Edges between F_d and S_f.** Consider (x, y) such that $\textsc{Find}(x) \in S_f$ and $y \in F_d$, then $k(\textsc{Find}(x)) < k(\textsc{Find}(y))$ after reordering. We now consider edges (x, y) such that $x \in F_d$ and $\textsc{Find}(y) \in S_f$. We examine two cases.

   (a) **When $V(x) \neq V(y)$.** In this case from Lemma 28, we conclude that $k(x_f) < k(y)$, thus contradicting the fact that $\textsc{Find}(y) \in S_f$.

   (b) **When $V(x) = V(y)$.** In this case, we know that $y \in F_a$, and therefore, $k(x_f) < k(\textsc{Find}(y))$, which contradicts the fact that $\textsc{Find}(y) \in S_f$.

5. **Edges between F_d and S_b.** Consider $x \in F_d$ and $y \in S_b$, $k(\textsc{Find}(x)) < k(\textsc{Find}(y))$ both before and after the reordering procedure.

6. **Edges between S_f and B_d.** Consider $x \in S_f$ and $y \in B_d$, $k(\textsc{Find}(x)) < k(\textsc{Find}(y))$ both before and after the reordering procedure.

7. **Edges between S_b and B_d.** Consider edge (x, y) with $x \in B_d$ and $\textsc{Find}(y) \in S_b$, $k(\textsc{Find}(x)) < k(\textsc{Find}(y))$ after reordering the vertices. We now show that there are no edges (x, y) such that $\textsc{Find}(x) \in S_b$ and $y \in B_d$.

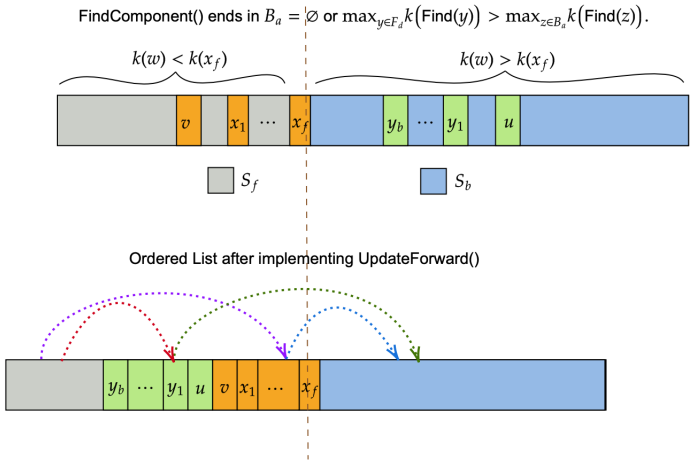

Figure 3: The algorithm UpdateForward places all vertices of F_d (in orange) and B_d (in green) immediately before x_f. The edges between S_f and F_d (in purple), between B_d and S_b (in green), between S_f and B_d (in red) and, between F_d and S_b (in blue) are consistent with the final ordering.

(a) **When** $V(x) = V(y)$. Assume there is such an edge, then in this case, $x \in B_a$ and $k(x_f) < k(\text{Find}(x)) < k(y_1)$. Since $B_a \neq \emptyset$, the algorithm terminates in the condition $\max_{z \in B_a} k(\text{Find}(z)) \leq \max_{w \in F_d} k(\text{Find}(w)) = k(x_f)$. On the other hand, $k(\text{Find}(x)) \leq \max_{z \in B_a} k(\text{Find}(z))$. This implies that $k(\text{Find}(x)) \leq k(x_f)$, which contradicts the fact that $\text{Find}(x) \in S_b$.

(b) **When** $V(x) \neq V(y)$**.** Since $V(y) = V(x_f)$, we conclude that $V(x_f) \neq V(x)$ as well. Additionally, since (x, y) is an edge, this implies that $V(x) \prec^* V(x_f)$, and by Lemma 28 we conclude that $k(\text{Find}(x)) < k(x_f)$, thus contradicting the fact that $\text{Find}(x) \in S_b$.

Theorem 57. The topological sort algorithm only has to place $O(|F_d|)$ vertices in their correct position in the ordered list. This takes $O(1)$ time per element. So, the total runtime is $O(\sqrt{mn\tau})$.

Total Runtime. Combining the runtimes of Phase 1, 2 and 3, we get a total runtime of $O(\sqrt{m^3\tau/n} + \sqrt{mn/\tau})$. Substituting $\tau = n/m^{1/3}$ we get the desired bound of $O(m^{4/3})$.