An Improved Algorithm for Incremental Cycle Detection and Topological Ordering in Sparse Graphs[*]

Sayan Bhattacharya[†] Janardhan Kulkarni[‡]

Abstract
We consider the problem of incremental cycle detection and topological ordering in a directed graph $G = (V, E)$ with $|V| = n$ nodes. In this setting, initially the edge-set E of the graph is empty. Subsequently, at each time-step an edge gets inserted into G. After every edge-insertion, we have to report if the current graph contains a cycle, and as long as the graph remains acyclic, we have to maintain a topological ordering of the node-set V. Let m be the total number of edges that get inserted into G. We present a randomized algorithm for this problem with $\tilde{O}(m^{4/3})$ total expected update time.

Our result improves the $\tilde{O}(m \cdot \min(m^{1/2}, n^{2/3}))$ total update time bound of [5, 9, 10, 7]. In particular, for $m = O(n)$, our result breaks the longstanding $\tilde{\Theta}(n^{3/2})$ barrier on the total update time. Furthermore, whenever $m = o(n^{3/2})$, our result improves upon the recently obtained $\tilde{O}(m\sqrt{n})$ total update time bound of [6]. We note that if $m = \Omega(n^{3/2})$, then the algorithm of [5, 4, 7], which has $\tilde{O}(n^2)$ total update time, beats the performance of the $\tilde{O}(m\sqrt{n})$ time algorithm of [6]. It follows that we improve upon the total update time of the algorithm of [6] in the "interesting" range of sparsity where $m = o(n^{3/2})$.

Our result also happens to be the first one that breaks the $\Omega(n\sqrt{m})$ lower bound of [9] on the total update time of any *local* algorithm for a nontrivial range of sparsity. Specifically, the total update time of our algorithm is $o(n\sqrt{m})$ whenever $m = o(n^{6/5})$. From a technical perspective, we obtain our result by combining the algorithm of [6] with the *balanced search* framework of [10].

1 Introduction

Consider an *incremental* directed graph $G = (V, E)$ with $|V| = n$ nodes. The edge-set E is empty in the beginning. Subsequently, at each time step an edge gets inserted into E. After each such *update* (edge insertion), we have to report if the current graph G contains a cycle, and as long as the graph remains acyclic, we have to maintain a topological ordering in G. The time taken to report the answer after an edge insertion is called the *update time*. We want to design an incremental algorithm for this problem with small *total update time*, which is defined as the sum of the update times over all the edge insertions. Recall that in the static setting there is an algorithm for cycle detection and topological ordering that runs in linear time. Thus, in the incremental setting, a naive approach would be to run this static algorithm from scratch after every edge-insertion in G. Let m be the number of edges in the final graph. Then the naive incremental algorithm will have a total update time of $O(m \times (m + n)) = O(m^2 + mn)$. In contrast, we get the result stated in Theorem 1.1. In Section 3, we also give instances which show that our analysis of the update time of the algorithm referred to in the theorem below is tight.

Theorem 1.1. *There is a randomized algorithm for incremental cycle detection with expected total update time of $\tilde{O}(m^{4/3})$.*

1.1 Perspective Cycle detection and topological ordering in directed graphs are fundamental, textbook problems. It is natural to ask what happens to the complexity of these problems when the input graph changes with time via a sequence of edge insertions. It comes as no surprise, therefore, that a long and influential line of work in the dynamic algorithms community, spanning over a couple of decades, have focussed on this question [6, 9, 10, 5, 4, 7, 2, 3, 12, 14, 15, 16].

Note that the problem is trivial in the *offline setting*. Here, we get an empty graph $G = (V, E)$ and a sequence of edges $e_1, \ldots, e_m$ as input *at one go*. For each $t \in [1, m]$, let G_t denote the status of G after the first t edges $e_1, \ldots, e_t$ have been inserted into E. We have to determine, for each t, if the graph G_t contains a cycle. This offline version can easily be solved in $O(m \log m)$ time using binary search. In contrast, we are still far away from designing an algorithm for the actual, incremental version of the problem that has $\tilde{O}(m)$ total update time.[1] This is especially relevant, because at present we do not know of any technique in the conditional lower bounds literature [1, 11, 13] that can prove a separation between the best possible total update time for an incremental problem and the

[*]A full version of the paper is available at https://arxiv.org/abs/1810.03491

[†]University of Warwick, UK.

[‡]Microsoft Research, Redmond, USA.

[1]Throughout this paper, we use the $\tilde{O}(.)$ notation to hide polylog factors.

best possible running time for the corresponding offline version. Thus, although it might be the case that there is no incremental algorithm for cycle detection and topological ordering with near-linear total update time, proving such a statement is beyond the scope of current techniques. With this observation in mind, we now review the current state of the art on the algorithmic front. We mention three results that are particularly relevant to this paper.

Result (1): There is an incremental algorithm with total update time of $\tilde{O}(n^2)$. This follows from the work of [5, 4, 7]. So the problem is well understood for dense graphs where $m = \Theta(n^2)$.

Result (2): There is an incremental algorithm with total update time of $\tilde{O}(m \cdot \min(m^{1/2}, n^{2/3}))$. This follows from the work of [5, 9, 10, 7].

Result (3): There is a randomized incremental algorithm with total expected update time of $\tilde{O}(m\sqrt{n})$. This follows from the very recent work of [6].

Significance of Theorem 1.1. We obtain a randomized incremental algorithm for cycle detection and topological ordering that has an expected total update time of $\tilde{O}(m^{4/3})$. Prior to this, all incremental algorithms for this problem had a total update time of $\Omega(n^{3/2})$ for sparse graphs with $m = \Theta(n)$. Our algorithm breaks this barrier by achieving a bound of $\tilde{O}(n^{4/3})$ on sparse graphs. More generally, our total update time bound of $\tilde{O}(m^{4/3})$ outperforms the $\tilde{O}(m\sqrt{n})$ bound from result (3) as long as $m = o(n^{3/2})$. Note that if $m = \omega(n^{3/2})$ then result (3) gets superseded by result (1). On the other hand, result (3) is no worse than result (2) for all values of m.[2][3] Thus, prior to our work result (3) gave the best known total update time when $m = o(n^{3/2})$, whereas result (1) gave the best known total update time when $m = \Omega(n^{3/2})$. We now improve upon the bound from result (3) in this "interesting" range of sparsity where $m = o(n^{3/2})$.

We are also able to break, for the first time in the literature, a barrier on the total update time of a certain type of algorithms that was identified by Haeupler et al. [10]. Specifically, they defined an algorithm to be *local* iff it satisfies the following property. Suppose that currently the graph G is acyclic, and the algorithm maintains a topological ordering $\prec$ on the node-set V such that $x \prec y$ for every edge $(x, y) \in E$. In other words, every edge is a *forward edge* under $\prec$. At this point, a directed edge (u, v) gets inserted into the graph G. Then the algorithm updates the topological ordering after this edge insertion only if $v \prec u$. Furthermore, if $v \prec u$, then the algorithm changes the positions of only those nodes in this topological ordering that lie in the *affected region*, meaning that a node x changes its position only if $v \preceq x \preceq u$ just before the insertion of the edge. Haeupler et al. [10] showed that any local algorithm for incremental cycle detection and topological ordering must necessarily have a total update time of $\Omega(n\sqrt{m})$. Interestingly, although the algorithms that lead to results (1) and (3) are *not* local, prior to our work no algorithm (local or not) was known in the literature that beats this $\Omega(n\sqrt{m})$ lower bound for *any* nontrivial value of m. In sharp contrast, our algorithm (which is *not* local) has a total update time of $\tilde{O}(m^{4/3})$, and this beats the $\Omega(n\sqrt{m})$ lower bound of Haeupler et al. [10] when $m = o(n^{6/5})$.

Our Technique. We obtain our result by combining the framework of Bernstein and Chechik [6] with the balanced search procedure of Haeupler et al. [10]. We first present a high level overview of the algorithm in [6]. Say that a node x is an ancestor (resp. descendant) of another node y iff there is a directed path from x to y (resp. from y to x) in the current graph G. The algorithm in [6] is parameterized by an integer $\tau \in [1, n]$ whose value will be fixed later on. Initially, each node $v \in V$ is sampled with probability $\Theta(\log n/\tau)$. Bernstein and Chechik [6] maintain a partition of the node-set V into subsets $\{V_{i,j}\}$, where a node v belongs to a subset $V_{i,j}$ iff it has exactly i ancestors and j descendants among the sampled nodes. A total order $\prec^*$ is defined on the subsets $\{V_{i,j}\}$, where $V_{i,j} \prec^* V_{i',j'}$ iff either $i < i'$ or $\{i = i', j > j'\}$. Next, it is shown that this partition and the total order satisfies two important properties. (1) If G contains a cycle, then all the nodes in that cycle belong to the same subset in the partition. (2) As long as G remains acyclic, every edge $(u, v) \in E$ is either an *internal edge* or a *forward edge* w.r.t. the total order $\prec^*$; this means that the subset containing u is either the same as or appears before the subset containing v. Intuitively, these two properties allow us to *decompose* the problem into smaller parts. All we need to do now is (a) maintain the subgraphs G_{ij} induced by the subsets V_{ij}, and (b) maintain a topological ordering within each subgraph $G_{i,j}$. Task (a) is implemented by using an incremental algorithm for single-source reachability and a data structure for maintaining an *ordered list* [8].

[2]Throughout this paper we assume that $m \geq n$. This is because if $m = o(n)$ then many nodes remain isolated (with zero degree) in the final graph, and we can ignore those isolated nodes while analyzing the total update time of the concerned algorithm.

[3]It is easy to combine two incremental algorithms and get the "best of both worlds". For example, suppose that we want to combine results (1) and (3) to get a total update time of $\tilde{O}(\min(n^2, m\sqrt{n}))$, without knowing the value of m in advance. Then we can initially start with the algorithm from result (3) and then *switch to* the algorithm from result (1) when m becomes $\Omega(n^{3/2})$.

For task (b), consider the scenario where an edge (u, v) gets inserted and both u and v belong to the same subgraph $G_{i,j}$. Suppose that u appears after v in the current topological ordering in $G_{i,j}$. We now have to check if the insertion of the edge (u, v) creates a cycle, or, equivalently, if there already exists a directed path from v to u. In [6] this task is performed by doing a *forward search* from v. Intuitively, this means exploring the nodes that are reachable from v and appear before u in the current topological ordering. If we encounter the node u during this forward search, then we have found the desired path from v to u, and we can report that the insertion of the edge (u, v) indeed creates a cycle. The time taken to implement this forward search is determined by the number of nodes x that are explored during this search. Bernstein and Chechik [6] now introduce a crucial notion of *τ-related pairs of nodes* (see Section 2.1 for details), and show that for every node x explored during the forward search we get a newly created τ-related pair (x, u). Next, they prove an upper bound of $O(n\tau)$ on the total number of such pairs that can appear throughout the duration of the algorithm. This implies that the total number of nodes explored during forward search is also at most $O(n\tau)$, and this in turn helps us fix the value of τ (to balance the time taken for task (a)) and bound the total update time.

We now explain our main idea. Inspired by the balanced search technique from [10], we modify the subroutine for implementing task (b) as follows. We simultaneously perform a *forward search* from v *and* a *backward search* from u. The forward search proceeds as in [6]. The backward search, on the other hand, explores the nodes y such that u is reachable from y and y appears after v in the current topological ordering. We alternate between a forward search step and a backward search step, so that at any point in time the number of nodes respectively explored by these two searches are equal to one other. If these two searches *meet* at some node z, then we have found a path from v to u (the path goes via z), and we accordingly declare that the insertion of the edge (u, v) creates a cycle. The time taken to implement task (b) is again determined by the number of nodes explored during the forward search, since this is the same as the number of nodes explored during the backward search. Now comes the following crucial observation. For every node x explored during the forward search and every node y explored during the backward search after the insertion of an edge (u, v), we get a newly created τ-related pair (x, y). Thus, if λ nodes are explored by each of these searches, then we get $\Omega(\lambda^2)$ newly created τ-related pairs; although we still explore only 2λ nodes overall. In contrast, the algorithm in [6] creates only $O(\lambda)$ many new τ-related pairs whenever it explores λ nodes. This *quadratic improvement* in the creation of new τ-related pairs leads to a much stronger bound on the total number of nodes explored by our algorithm, because as in [6] we still can have at most $O(n\tau)$ many newly created τ-related pairs during the entire course of the algorithm. This improved bound on the number of explored nodes leads to an improved bound of $\tilde{O}(m^{4/3})$ on the total update time.

2 Our Algorithm: Proof of Theorem 1.1

This section is organized as follows. In Section 2.1 we recall some useful concepts from [6]. In Section 2.2 we present our incremental algorithm, and in Section 2.3 we analyze its total update time.

2.1 Preliminaries Throughout the paper, we assume that the maximum degree of a node in G is at most $O(1)$ times the average degree. It was observed in [6] that this assumption is without any loss of generality.

Assumption 1. *[6] Every node in G has an out-degree of $O(m/n)$ and an in-degree of $O(m/n)$.*

We say that a node $x \in V$ is an *ancestor* of another node $y \in V$ iff there is a directed path from x to y in G. We let $A(y) \subseteq V$ denote the set of all ancestors of $y \in V$. Similarly, we say that x is a *descendant* of y iff there is a directed path from y to x in G. We let $D(y) \subseteq V$ denote the set of all descendants of y. A node is both an ancestor and a descendant of itself, that is, we have $x \in A(x) \cap D(x)$. We also fix an integral parameter $\tau \in [1, n]$ whose exact value will be determined later on. Note that if there is a path from a node x to another node y in G, then $A(x) \subseteq A(y)$ and $D(y) \subseteq D(x)$. Such a pair of nodes is said to be *τ-related* iff the number of nodes in each of the sets $A(y) \setminus A(x)$ and $D(x) \setminus D(y)$ does not exceed τ.

Definition 2.1. *[6] We say that an ordered pair of nodes (x, y) is* τ-related *in the graph G iff there is a path from x to y in G, and $|A(y) \setminus A(x)| \leq \tau$ and $|D(x) \setminus D(y)| \leq \tau$. We emphasize that for the ordered pair (x, y) to be τ-related, it is* not *necessary that there be an edge $(x, y) \in E$.*

If two nodes $x, y \in V$ are part of a cycle, then clearly $A(x) = A(y)$ and $D(x) = D(y)$, and both the ordered pairs (x, y) and (y, x) are τ-related. In other words, if an ordered pair (x, y) is not τ-related, then there is no cycle containing both x and y. Intuitively, therefore, the notion of τ-relatedness serves as a *relaxation* of the notion of two nodes being part of a cycle. Next, note that the graph G keeps changing as more and more edges are inserted into it. So it might be the case that an ordered pair of nodes (x, y) is *not* τ-related in G at some point in time, but *is* τ-related in G at some other point in time. The following definition and the subsequent theorem becomes relevant in light of this observation.

Definition 2.2. *[6] We say that an* ordered *pair of nodes (x, y) is* sometime τ-related *in the graph G iff it is τ-related at some point in time during the entire sequence of edge insertions in G.*

Theorem 2.1. *[6] The number of sometime τ-related pairs of nodes in G is at most $O(n\tau)$.*

Following [6], we maintain a partition of the node-set V into subsets $\{V_{i,j}\}$ and the subgraphs $\{G_{i,j} = (V_{i,j}, E_{i,j})\}$ induced by these subsets of nodes. We sample each node $x \in V$ independently with probability $\log n/\tau$. Let $S \subseteq V$ denote the set of these sampled nodes. The outcome of this random sampling gives rise to a partition of the node-set V into $(|S| + 1)^2$ many subsets $\{V_{i,j}\}$, where $i, j \in [0, |S|]$. This is formally defined as follows. For every node $x \in V$, let $A_S(x) = A(x) \cap S$ and $D_S(x) = D(x) \cap S$ respectively denote the set of ancestors and descendants of x that have been sampled. Each subset $V_{i,j} \subseteq V$ is indexed by an ordered pair (i, j) where $i \in [0, |S|]$ and $j \in [0, |S|]$. A node $x \in V$ belongs to a subset $V_{i,j}$ iff $|A_S(x)| = i$ and $|D_S(x)| = j$. In words, the index (i, j) of the subset $V_{i,j}$ specifies the number of sampled ancestors and sampled descendants each node $x \in V_{i,j}$ is allowed to have. It is easy to check that the subsets $\{V_{i,j}\}$ form a valid partition of the node-set V. Let $E_{i,j} = \{(x, y) \in E : x, y \in V_{i,j}\}$ denote the set of edges in G whose both endpoints lie in $V_{i,j}$, and let $G_{i,j} = (V_{i,j}, E_{i,j})$ denote the subgraph of G induced by the subset of nodes $V_{i,j}$. We also define a total order $\prec^*$ on the subsets $\{V_{i,j}\}$, where we have $V_{i,j} \prec^* V_{i',j'}$ iff either $\{i < i'\}$ or $\{i = i', j > j'\}$. We slightly abuse the notation by letting $V(x)$ denote the unique subset $V_{i,j}$ which contains the node $x \in V$. Consider any edge $(x, y) \in E$. If the two endpoints of the edge belong to two different subsets in the partition $\{V_{i,j}\}$, i.e., if $V(x) \neq V(y)$, then we refer to the edge (x, y) as a *cross edge*. Otherwise, if $V(x) = V(y)$, then the edge (x, y) is an *internal edge*.

Lemma 2.1. *[6] Consider the partition of the node-set V into subsets $\{V_{i,j}\}$, and the subgraphs $\{G_{i,j} = (V_{i,j}, E_{i,j})\}$ induced by these subsets of nodes. They satisfy the following three properties.*

- *If there is a cycle in $G = (V, E)$, then every edge of that cycle is an internal edge.*
- *For every cross edge $(x, y) \in E$, we have $V(x) \prec^* V(y)$.*
- *Consider any two nodes $x, y \in V_{i,j}$ for some $i, j \in [0, |S|]$. If there is a path from x to y in the subgraph $G_{i,j}$, then with high probability the ordered pair (x, y) is τ-related in G.*

The first property states that the graph G contains a cycle iff some subgraph $G_{i,j}$ contains a cycle. Hence, in order to detect a cycle in G it suffices to only consider the edges that belong to the induced subgraphs $\{G_{i,j}\}$. The second property, on the the other hand, implies that if the graph G is acyclic, then it admits a topological ordering $\prec$ that is *consistent* with the total order $\prec^*$, meaning that $x \prec y$ for all $x, y \in V$ with $V(x) \prec^* V(y)$. Finally, the last property states that whenever a subgraph $G_{i,j}$ contains a path from a node x to some other node y, with high probability the ordered pair (x, y) is τ-related in the input graph G.

2.2 The algorithm

Since edges never get deleted from the graph G, our algorithm does not have to do anything once it detects a cycle (for the graph will continue to have a cycle after every edge-insertion in the future). Accordingly, we assume that the graph G has remained acyclic throughout the sequence of edge insertions untill the present moment, and our goal is to check if the next edge-insertion creates a cycle in G. Our algorithm maintains a topological ordering $\prec$ of the node-set V in the graph G that is *consistent* with the total order $\prec^*$ on the subsets of nodes $\{V_{i,j}\}$, as defined in Section 2.2. Specifically, we maintain a *priority* $k(x)$ for every node $x \in V$, and for every two nodes $x, y \in V$ with $V(x) \prec^* V(y)$ we ensure that $k(x) \prec k(y)$. As long as G remains acyclic, the existence of such a topological ordering $\prec$ is guaranteed by Lemma 2.1.

Data Structures. We maintain the partition $\{V_{i,j}\}$ of the node-set V and the subgraphs $\{G_{i,j} = (V_{i,j}, E_{i,j})\}$ induced by the subsets in this partition. We use an *ordered list* data structure [8] on the node-set V to implicitly maintain the priorities $\{k(x)\}$ associated with the topological ordering $\prec$. This data structure supports each of the following operations in $O(1)$ time.

- INSERT-BEFORE(x, y): This inserts the node y just before the node x in the topological ordering.
- INSERT-AFTER(x, y): This inserts the node y just after the node x in the topological ordering.
- DELETE(x): This deletes the node x from the existing topological ordering.
- COMPARE(x, y): If $k(x) \prec k(y)$, then this returns YES, otherwise this returns NO.

The implementation of our algorithm requires the creation of two *dummy nodes* $x_{i,j}$ and $y_{i,j}$ in every subset $V_{i,j}$. We ensure that $k(x_{i,j}) \prec k(x) \prec k(y_{i,j})$ for all $x \in V_{i,j}$. In words, the dummy node $x_{i,j}$ (resp. $y_{i,j}$) comes *first* (resp. *last*) in the topological order among all the nodes in $V_{i,j}$. Further, for all nodes $x \in V$ with $V(x) \prec V_{i,j}$ we have $k(x) \prec k(x_{i,j})$, and for all nodes $x \in V$ with $V_{i,j} \prec V(x)$ we have $k(y_{i,j}) \prec k(x)$.

Handling the insertion of an edge (u, v) in G. By induction hypothesis, suppose that the graph G currently does not contain any cycle and we are maintaining the topological ordering $\prec$ in G. At this point, an edge (u, v) gets inserted into G. Our task now is to first figure out if the insertion of this edge creates a cycle, and if not, then to update the topological ordering $\prec$. We perform this task in four *phases*, as described below.

1. In phase I, we update the subgraphs $\{G_{i,j}\}$.
2. In phase II, we update the total order $\prec$ to make it consistent with the total order $\prec^*$.
3. In phase III, we check if the edge-insertion creates a cycle in G. See Section 2.2.1 for details.
4. If phase III fails to detect a cycle, then in phase IV we further update (if necessary) the total order $\prec$ so as to ensure that it is a topological order in the current graph G. See Section 2.2.2 for details.

Remark. We follow the framework developed in [6] while implementing Phase I and Phase II. We differ from [6] in Phase III and Phase IV, where we use the *balanced search* approach from [10].

Implementing Phase I. In the first phase, we update the subgraphs $\{G_{i,j}\}$ such that they satisfy the properties mentioned in Lemma 2.1. The next lemma follows from [6]. The key idea is to maintain incremental single-source reachability data structures from each of the sampled nodes. Since at most $\tilde{O}(n/\tau)$ many nodes are sampled in expectation, and since each incremental single-source reachability data structure requires $\tilde{O}(m)$ total update time to handle m edge insertions, we get the desired bound of $\tilde{O}(mn/\tau)$.

Lemma 2.2. *[6] In phase I, the algorithm spends $\tilde{O}(mn/\tau)$ total update time in expectation.*

Implementing Phase II. In this phase we update the total order $\prec$ on the node-set V in a certain manner. Let G^- and G^+ respectively denote the graph G just before and just after the insertion of the edge (u, v). Similarly, for every node $x \in V$, let $V^-(x)$ and $V^+(x)$ respectively denote the subset $V(x)$ just before and just after the insertion of the edge (u, v). At the end of this phase, the following properties are satisfied.

Property 2.1. *[6] At the end of phase II the total order $\prec$ on V is consistent with the total order $\prec^*$ on $\{V_{i,j}\}$. Specifically, for any two nodes x and y, if $V(x) \prec^* V(y)$, then we also have $k(x) \prec k(y)$.*

Property 2.2. *[6] At the end of phase II the total order $\prec$ on V remains a valid topological ordering of G^-, where G^- denotes the graph G just before the insertion of the edge (u, v).*

The next lemma bounds the total time spent by the algorithm in phase II.

Lemma 2.3. *[6] The total time spent in phase II across all edge-insertions is at most $\tilde{O}(n^2/\tau)$.*

Proof. (Sketch) Let C be a counter that keeps track of the number of times some node moves from one subset in the partition $\{V_{i,j}\}$ to another. Recall that a node $x \in V$ belongs to a subset $V_{i,j}$ iff $|A_S(x)| = i$ and $|D_S(x)| = j$. As more and more edges keep getting inserted in G, the node x can never lose a sampled node in S as its ancestor or descendent. Instead, both the sets $A_S(x)$ and $D_S(x)$ can only grow with the passage of time. Since $|A_S(v)|, |D_S(v)| \in [0, |S|]$, each node x can move from one subset in the partition $\{V_{i,j}\}$ to another at most $2 \cdot |S|$ times. Thus, we have $C \leq |V| \cdot 2|S| = O(n|S|)$. Since $\mathbf{E}[|S|] = \tilde{O}(n/\tau)$, we conclude that $\mathbf{E}[C] = \tilde{O}(n^2/\tau)$. Now, phase II can be implemented in such a way that a call is made to the ordered list data structure [8] only when some node moves from one subset of the partition $\{V_{i,j}\}$ to another. So the total time spent in phase II is at most C, which happens to be $\tilde{O}(n^2/\tau)$ in expectation. ∎

2.2.1 Phase III: Checking if the insertion of the edge (u, v) creates a cycle.

Let G^- and G^+ respectively denote the graph G before and after the insertion of the edge (u, v). Consider the total order $\prec$ on the set of nodes V in the beginning of phase III (or, equivalently, at the end of phase II). Property 2.1 guarantees that $\prec$ is consistent with the total order $\prec^*$ on $\{V_{i,j}\}$, and Property 2.2 guarantees that $\prec$ is a valid topological ordering in G^-. We will use these two properties throughout the current phase. The pseudocodes of all the subroutines used in this phase appear in Section 2.4.

In phase III, our goal is to determine if the insertion of the edge (u, v) creates a cycle in G. Note that if $k(u) \prec k(v)$, then $\prec$ is also a valid topological ordering in G^+ as per Property 2.2, and clearly the insertion of the edge (u, v) does not create a cycle. The difficult case occurs when $k(v) \prec k(u)$. In this case, we first infer that $V(u) = V(v)$, meaning that both u and v

belong to the same subset in the partition $\{V_{i,j}\}$ at the end of phase II. This is because of the following reason. The total order $\prec$ is consistent with the total order $\prec^*$ as per Property 2.1. Accordingly, since $k(v) \prec k(u)$, we conclude that if $V(v) \neq V(u)$ then $V(v) \prec^* V(u)$. But this would contradict Lemma 2.1 as there is a cross edge from u to v.

To summarize, for the rest of this section we assume that $k(v) \prec k(u)$ and $V(v) = V(u) = V_{i,j}$ for some $i, j \in [0, |S|]$. We have to check if there is a path $P_{v,u}$ from v to u in G^-. Along with the edge (u, v), such a path $P_{v,u}$ will define a cycle in G^+. Hence, by Lemma 2.1, every edge e in such a path $P_{v,u}$ will belong to the subgraph $G_{i,j} = (V_{i,j}, E_{i,j})$. Thus, from now on our task is to determine if there is a path $P_{v,u}$ from v to u in $G_{i,j}$. We perform this task by calling the subroutine SEARCH(u, v) described below.

SEARCH(u, v). We conduct two searches in order to find the path $P_{v,u}$: A *forward search* from v, and a *backward search* from u. Specifically, let F and B respectively denote the set of nodes visited by the forward search and the backward search untill now. We always ensure that $F \cap B = \emptyset$. A node in F (resp. B) is referred to as a forward (resp. backward) node. Every forward node $x \in F$ is reachable from the node v in $G_{i,j}$, whereas the node u is reachable from every backward node $x \in B$ in $G_{i,j}$. We further classify each of the sets F and B into two subsets: $F_a \subseteq F$, $F_d = F \setminus F_a$ and $B_a \subseteq B$, $B_d = B \setminus B_a$. The nodes in F_a and B_a are called *alive*, whereas the nodes in F_d and B_d are called *dead*. Intuitively, the dead nodes have already been *explored* by the search, whereas the alive nodes have not yet been explored. When the subroutine begins execution, we have $F_a = \{v\}$ and $B_a = \{u\}$. The following property is always satisfied.

PROPERTY 2.3. *Every node $x \in F_a \cup F_d$ is reachable from the node v in $G_{i,j}$, and the node u is reachable from every node $y \in B_a \cup B_d$ in $G_{i,j}$. The sets F_a, F_d, B_a and B_d are pairwise mutually exclusive.*

A simple strategy for exploring a forward and alive node $x \in F_a$ is as follows. For each of its outgoing edges $(x, y) \in E_{i,j}$, we check if $y \in B$. If yes, then we have detected a path from v to u: This path goes from v to x (this is possible since x is a forward node), follows the edge (x, y), and then from y it goes to u (this is possible since y is a backward node). Accordingly, we stop and report that the graph G^+ contains a cycle. In contrast, if $y \notin B$ and $y \notin F$, then we insert y into the set F_a (and F), so that y becomes a forward and alive node which will be explored in future. In the end, we move the node x from the set F_a to the set F_d. We refer to the subroutine that explores a node $x \in F_a$ as **EXPLORE-FORWARD(x)**.

Analogously, we explore a backward and alive node $x \in B_a$ is as follows. For each of its incoming edges $(y, x) \in E_{i,j}$, we check if $y \in F$. If yes, then there is a path from v to u: This path goes from v to y (this is possible since y is a forward node), follows the edge (y, x), and then from x it goes to u (this is possible since x is a backward node). Accordingly, we stop and report that the graph G^+ contains a cycle. In contrast, if $y \notin F$ and $y \notin B$, then we insert y into the set B_a (and B), so that y becomes a backward and alive node which will be explored in future. In the end, we move the node x from the set B_a to the set B_d. We refer to the subroutine that explores a node $x \in B_a$ as **EXPLORE-BACKWARD(x)**.

PROPERTY 2.4. *Once a node $x \in F_a$ (resp. $x \in B_a$) has been explored, we delete it from the set F_a (resp. B_a) and insert it into the set F_d (resp. B_d).*

While exploring a node $x \in F_a$ (resp. $x \in B_a$), we ensure that all its outgoing (resp. incoming) neighbors are included in F (resp. B). This leads to the following important corollary.

COROLLARY 2.1. *Consider any edge $(x, y) \in E_{i,j}$. At any point in time, if $x \in F_d$, then at that time we also have $y \in F_a \cup F_d$. Similarly, at any point in time, if $y \in B_d$, then at that time we also have $x \in B_a \cup B_d$.*

Two natural questions arise at this point. First, how frequently do we explore forward nodes compared to exploring backward nodes? Second, suppose that we are going to explore a forward (resp. backward) node at the present moment. Then how do we select the node x from the set F_a (resp. B_a) that has to be explored? Below, we state two crucial properties of our algorithm that address these two questions.

PROPERTY 2.5. *(Balanced Search) We alternate between calls to EXPLORE-FORWARD(.) and EXPLORE-BACKWARD(.). This ensures that $|B_d| - 1 \leq |F_d| \leq |B_d| + 1$ at every point in time. In other words, every forward-exploration step is followed by a backward-exploration step and vice versa.*

PROPERTY 2.6. *(Ordered Search) While deciding which node in F_a to explore next, we always pick the node $x \in F_a$ that has* minimum *priority $k(x)$. Thus, we ensure that the subroutine EXPLORE-FORWARD(x) is only called on the node x that appears before every other node in F_a in the total ordering $\prec$. In contrast, while deciding which node in B_a to explore next, we always pick the node $y \in B_a$ that has* maximum *priority $k(y)$. Thus, we* *ensure that the subroutine EXPLORE-BACKWARD(y) is only called on the node x that appears* after *every other node in B_a in the total ordering $\prec$.*

An immediate consequence of Property 2.6 is that there is no *gap* in the set F_d as far as reachability from the node v is concerned. To be more specific, consider the sequence of nodes in $G_{i,j}$ that are reachable from v in increasing order of their positions in the total order $\prec$. This sequence starts with v. The set of nodes belonging to F_d always form a prefix of this sequence. This observation is formally stated below.

Corollary 2.2. *Consider any two nodes $x, y \in V_{i,j}$ such that $k(x) \prec k(y)$ and there is a path in $G_{i,j}$ from v to each of these two nodes. At any point in time, if $y \in F_d$, then we must also have $x \in F_d$.*

Corollary 2.3 is a mirror image of Corollary 2.2, albeit from the perspective of the node u.

Corollary 2.3. *Consider any two nodes $x, y \in V_{i,j}$ such that $k(x) \prec k(y)$ and there is a path in $G_{i,j}$ from each of these two nodes to u. At any point in time, if $x \in B_d$, then we must also have $y \in B_d$.*

To complete the description of the subroutine SEARCH(u, v), we now specify six *terminating conditions.* Whenever one of these conditions is satisfied, the subroutine does not need to run any further because it already knows whether or not the insertion of the edge (u, v) creates a cycle in the graph G.

(C1) $F_a = \emptyset$.

In this case, we conclude that the graph G remains acyclic even after the insertion of the edge (u, v). We now justify this conclusion. Recall that if the insertion of the edge (u, v) creates a cycle, then that cycle must contain a path $P_{v,u}$ from v to u in $G_{i,j}$. When the subroutine SEARCH(u, v) begins execution, we have $F_a = \{v\}$ and $B_a = \{u\}$. Hence, Property 2.4 implies that at the present moment $v \in F_d \cup F_a$ and $u \in B_d \cup B_a$. Since the sets F_d, F_a, B_d, B_a are pairwise mutually exclusive (see Property 2.3) and $F_a = \emptyset$, we currently have $v \in F_d$ and $u \notin F_d$. Armed with this observation, we consider the path P_{vu} from v to u, and let x be the first node in this path that does not belong to F_d. Let y denote the node that appears just before x in this path. Then by definition, we have $y \in F_d$ and $(y, x) \in E_{i,j}$. Now, applying Corollary 2.1, we get $x \in F_d \cup F_a = F_d$, which leads to a contradiction.

(C2) $B_a = \emptyset$.

This is analogous to the condition (C1) above, and we conclude that G remains acyclic in this case.

(C3) *While exploring a node $x \in F_a$, we discover that x has an outgoing edge to a node $x' \in B_a \cup B_d$.*

Here, we conclude that the insertion of the edge (u, v) creates a cycle. We now justify this conclusion. Since $x \in F_a$, Property 2.3 implies that there is a path $P_{v,x}$ from v to x. Since $x' \in B_a \cup B_d$, Property 2.3 also implies that there is a path $P_{x',u}$ from x' to u. We get a cycle by combining the path $P_{v,x}$, the edge (x, x'), the path $P_{x',u}$ and the edge (u, v).

(C4) *While exploring a node $y \in B_a$, we discover that y has an incoming edge from a node $y' \in F_a \cup F_d$.*

Similar to condition (C3), in this case we conclude that the insertion of the edge (u, v) creates a cycle.

(C5) $\min_{x \in F_a} k(x) \succ \min_{y \in B_d} k(y)$.

If this happens, then we conclude that the graph G remains acyclic even after the insertion of the edge (u, v). We now justify this conclusion. Suppose that the insertion of the edge (u, v) creates a cycle. Such a cycle defines a path $P_{v,u}$ from v to u. Below, we make a claim that will be proved later on.

Claim 1. *The path $P_{v,u}$ contains at least one node x from the set F_a.*

Armed with Claim 1, we consider any node x' in the path $P_{v,u}$ that belongs to the set F_a. Let $y' = \arg\min_{y \in B_d}\{k(y)\}$. Note that $k(y') = \min_{y \in B_a} k(y) \prec \min_{x \in F_a} k(x) \preceq k(x')$. In particular, we infer that $k(y') \prec k(x')$. As $y' \in B_d$, the node u is reachable from y' (see Property 2.3). Similarly, as the node x' lies on the path $P_{v,u}$, the node u is also reachable from x'. Since the node u is reachable from both the nodes $y' \in B_d$ and x', and since $k(y') \prec k(x)$, Corollary 2.3 implies that $x' \in B_d$. This leads to a contradiction, for $x' \in F_a$ and $F_a \cap B_d = \emptyset$ (see Property 2.3). Hence, our initial assumption was wrong, and the insertion of the edge (u, v) does not create a cycle in G. It now remains to prove Claim 1.

Proof of Claim 1. Applying the same argument used to justify condition (C1), we first observation that $v \in F_a \cup F_d$ and $u \in B_a \cup B_d$. As the subsets F_a, F_d, B_a and B_d are pairwise mutually exclusive (see Property 2.3), we have $u \notin F_a \cup F_d$. Note that if $v \in F_a$, then there is nothing further to prove. Accordingly, for the rest of the proof we consider the scenario where $v \in F_d$. Since $v \in F_d$ and $u \notin F_d$, there has to be at least one node in the path $P_{v,u}$ that does not belong to the set F_d. Let x be the first such node, and let y be the node that appears just before x in the path $P_{v,u}$. Thus, we have $y \in F_d$, $x \notin F_d$ and $(y, x) \in E_{i,j}$. Hence, Corollary 2.1 implies that $x \in F_a$. So the path $P_{v,u}$ contains some node from the set F_a.

(C6) $\max_{y \in B_a} k(y) \prec \max_{x \in F_d} k(x)$.

Similar to condition (C5), here we conclude that the graph G remains acyclic.

We now state an important corollary that follows from our stopping conditions (C5) and (C6). It states that every node $x \in F_d$ appears before every node $y \in B_d$ in the total order $\prec$ in phase III.

COROLLARY 2.4. *We always have* $\max_{x \in F_d}\{k(x)\} \prec \min_{y \in B_d}\{k(y)\}$.

Proof. Suppose that the corollary is false. Note that initially when the subroutine SEARCH(u, v) begins execution, we have $F_d = B_d = \emptyset$ and hence the corollary is vacuously true at that time. Consider the first time-instant (say) t when the corollary becomes false. Accordingly, we have:

$$\max_{x \in F_d}\{k(x)\} \prec \min_{y \in B_d}\{k(y)\} \text{ just before time } t. \tag{2.1}$$

One of the following two events must have occurred at time t for the corollary to get violated.

(1) A node $x' \in F_a$ was explored during a call to the subroutine EXPLORE-FORWARD(x'). The subroutine EXPLORE-FORWARD(x') then moved the node x' from the set F_a to the set F_d, which violated the corollary. Note that a call to EXPLORE-FORWARD(.) can only be made if $k(x') \prec \min_{y \in B_d}\{k(y)\}$ just before time t (see stopping condition (C5) and Property 2.6). Thus, from (2.1) we conclude that the corollary remains satisfied even after adding the node x' to the set F_d. This leads to a contradiction.

(2) A node $y' \in B_a$ was explored during a call to EXPLORE-BACKWARD(x'). The subroutine EXPLORE-BACKWARD(x') then moved the node y' from the set B_a to the set B_d, which violated the corollary. Applying an argument analogous to the one applied in case (1), we again reach a contradiction. □

The proof of Lemma 2.4 follows immediately from the preceding discussion. Next, Lemma 2.5 bounds the time spent in any single call to the subroutine SEARCH(u, v).

LEMMA 2.4. *The subroutine SEARCH(u, v) in Figure 1 returns YES if the insertion of the edge (u, v) creates a cycle in the graph G, and NO otherwise.*

LEMMA 2.5. *Consider any call to the subroutine SEARCH(u, v). The time spent on this call is at most $\tilde{O}(m/n)$ times the size of the set F_d at the end of the call.*

Proof. (Sketch) Each call to EXPLORE-FORWARD(x) or EXPLORE-BACKWARD(x) takes time proportional to the out-degree (resp. in-degree) of x in the subgraph $G_{i,j}$. Under Assumption 1, the maximum in-degree and maximum out-degree of a node in $G_{i,j}$ are both at most $O(m/n)$. Thus, a single call to EXPLORE-FORWARD(x) or EXPLORE-BACKWARD(x) takes $O(m/n)$ time.

According to Property 2.6, whenever we want to explore a node during forward-search (resp. backward-search), we select a forward-alive (resp. backward-alive) node with minimum (resp. maximum) priority. This step can be implemented using a priority queue data structure in $\tilde{O}(1)$ time.

So the time spent by procedure SEARCH(u, v) is at most $\tilde{O}(m/n)$ times the number of calls to the subroutines EXPLORE-FORWAD(.) or EXPLORE-BACKWARD(.). Furthermore, after each call to the subroutine EXPLORE-FORWAD(.) or EXPLORE-BACKWARD(.), the size of the set F_d or B_d respectively increases by one. Accordingly, the time spent on one call to SEARCH(u, v) is at most $\tilde{O}(m/n)$ times the size of the set $F_d \cup B_d$ at the end of the call. The lemma now follows from Property 2.5. □

Total time spent in phase III. We now analyze the total time spent in phase III, over the entire sequence of edge insertions in G. For $l \in [1, m]$, consider the l^{th} edge-insertion in the graph G, and let t_l denote the size of the set F_d at the end of phase III while handling this l^{th}-edge insertion. Lemma 2.5 implies that the total time spent in phase III is at most $\tilde{O}\left((m/n) \cdot \sum_{l=1}^{m} t_l\right)$. We now focus on upper bounding the sum $\sum_{l=1}^{m} t_l$.

LEMMA 2.6. *We have* $\sum_{l=1}^{m} t_l^2 = O(n\tau)$.

Proof. For any $l \in [1, m]$, let $F_d^{(l)}$ and $B_d^{(l)}$ respectively denote the sets F_d and B_d at the end of phase III while handling the l^{th} edge-insertion in G. Furthermore, let $G^{(l)}$ and $G_{i,j}^{(l)}$ respectively denote the input graph G and the subgraph $G_{i,j}$ after the l^{th} edge-insertion in G.

Suppose that the edge (u, v) is the l^{th} edge to be inserted into G. We focus on the procedure for handling this edge insertion. During this procedure, if we find $k(u) \prec k(v)$ in the beginning of phase III, then our algorithm immediately declares that the insertion of the edge (u, v) does not create a cycle and moves on to phase IV. In such a scenario, we clearly have $F_d^{(l)} = B_d^{(l)} = \emptyset$ and hence $t_l = 0$. Accordingly, from now on we assume that $k(v) \prec k(u)$ in the beginning of phase III. Consider any two nodes $x \in F_d^{(l)}$ and $y \in B_d^{(l)}$. The nodes x and y belong to the same subgraph $G_{i,j}^{(l)}$. Property 2.3 guarantees that there is a path $P_{y,x}$ from y to x in $G_{i,j}^{(l)}$

– we can go from y to u, take the edge (u,v) and then go from v to x. Hence, by Lemma 2.1, the ordered pair (y,x) is τ-related in $G^{(l)}$ with high probability. We condition on this event for the rest of the proof. We now claim that there was no path from y to x in $G^{(l-1)}$: this is the graph G just before the l^{th} edge-insertion, or equivalently, just after the $(l-1)^{th}$ edge-insertion. To see why this claim is true, we recall Property 2.2. This property states that in the beginning of phase III (after the l^{th} edge-insertion) the total order $\prec$ on the node-set V is a topological order in the graph $G^{(l-1)}$. Since $y \in B_d^{(l)}$ and $x \in F_d^{(l)}$, Corollary 2.4 implies that x appears before y in the total order $\prec$ in phase III (after the l^{th} edge-insertion). From these last two observations, we conclude that there is no path from y to x in $G^{(l-1)}$. As edges only get inserted into G with the passage of time, this also implies that there is no path from y to x in the graph $G^{(l')}$, for all $l' < l$. Accordingly, the ordered pair (y,x) is *not* τ-related in the graph $G^{(l')}$ for any $l' < l$.

To summarize, for every node $x \in F_d^{(l)}$ and every node $y \in B_d^{(l)}$ the following conditions hold. (1) The ordered pair (y,x) is τ-related in the graph $G^{(l)}$. (2) For all $l' < l$, the ordered pair (y,x) is *not* τ-related in the graph $G^{(l')}$. Let C denote a counter which keeps track of the number of sometime τ-related pairs of nodes (see Definition 2.2). Conditions (1) and (2) imply that every ordered pair of nodes (y,x), where $y \in B_d^{(l)}$ and $x \in F_d^{(l)}$, contributes one towards the counter C. A simple counting argument gives us:

$$\sum_{l=1}^{m} \left|F_d^{(l)}\right| \cdot \left|B_d^{(l)}\right| \leq C = O(n\tau) \tag{2.2}$$

In the above derivation, the last equality follows from Theorem 2.1. We now recall Property 2.5, which says that our algorithm in phase III explores (almost) the same number of forward and backward nodes. In particular, we have $\left|F_d^{(l)}\right| \cdot \left|B_d^{(l)}\right| = O\left(\left|F_d^{(l)}\right|^2\right) = O(t_l^2)$ for all $l \in [1,m]$. This observation, along with (2.2), implies that $\sum_{l=1}^m t_l^2 = O(n\tau)$. This concludes the proof of the lemma. □

Corollary 2.5. *We have* $\sum_{l=1}^m t_l = O(\sqrt{mn\tau})$.

Proof. We partition the set of indices $\{1,\ldots,m\}$ into two subsets:

$$X = \left\{ l \in [1,m] : t_l \leq \sqrt{n\tau/m} \right\}.$$

$$Y = \left\{ l \in [1,m] : t_l > \sqrt{n\tau/m} \right\}.$$

It is easy to check that $\sum_{l \in X} t_l \leq |X| \cdot \sqrt{n\tau/m} \leq m \cdot \sqrt{n\tau/m} = \sqrt{mn\tau}$. Accordingly, for the rest of the proof we focus on bounding the sum $\sum_{l\in Y} t_l$. Towards this end, for each $l \in Y$, we first express the quantity t_l as $t_l = \sqrt{n\tau/m} + \delta_l$, where $\delta_l > 0$. Now, Lemma 2.6 implies that:

$$\sum_{l\in Y} t_l^2 = \sum_{l \in Y} \left(\sqrt{n\tau/m} + \delta_l\right)^2 = O(n\tau) \tag{2.3}$$

We also note that:

$$\begin{aligned} \sum_{l\in Y} \left(\sqrt{n\tau/m} + \delta_l\right)^2 &\geq \sum_{l\in Y} \left(\delta_l \cdot \sqrt{n\tau/m}\right) \\ &= \sqrt{n\tau/m} \cdot \sum_{l\in Y} \delta_l. \end{aligned} \tag{2.4}$$

From (2.3) and (2.4), we get $\sqrt{n\tau/m} \cdot \sum_{l\in Y} \delta_l = O(n\tau)$, which in turn gives us: $\sum_{l\in Y} \delta_l = O\left(\sqrt{mn\tau}\right)$. This leads to the following upper bound on the sum $\sum_{l\in Y} t_l$.

$$\begin{aligned} \sum_{l\in Y} t_l &= \sum_{l\in Y} \left(\sqrt{n\tau/m} + \delta_l\right) \\ &= \sum_{l\in Y} \sqrt{n\tau/m} + \sum_{l \in Y} \delta_l \\ &\leq m \cdot \sqrt{n\tau/m} + O\left(\sqrt{mn\tau}\right) \\ &= O\left(\sqrt{mn\tau}\right). \end{aligned}$$

This concludes the proof of the corollary. □

We are now ready to upper bound the total time spent by our algorithm in phase III.

Lemma 2.7. *We spend* $\tilde{O}\left(\sqrt{m^3\tau/n}\right)$ *total time in phase III, over the entire sequence of edge-insertions.*

Proof. Lemma 2.5 implies that the total time spent in phase III is $O\left((m/n) \cdot \sum_{l=1}^m t_l\right)$. The lemma now follows from Corollary 2.5. □

2.2.2 Phase IV: Ensuring that $\prec$ is a topological ordering for G^+ (only when G^+ is acyclic)

As in Section 2.2.1, we let G^- and G^+ respectively denote the graph G just before and after the insertion of the edge (u,v). If in phase III we detect a cycle, then we do not need to perform any nontrivial computation from this point onward, for the graph G will contain a cycle after every future edge-insertion. Hence, throughout this section we assume that no cycle was detected in phase III, and as per Lemma 2.4 the graph G^+ is acyclic. Our goal in phase IV is to update the total order $\prec$ so that it becomes a topological ordering in G^+. Towards this end, note that $\prec$ does not change during phase

III. Furthermore, if $k(u) \prec k(v)$ in phase III, then the first three paragraphs of Section 2.2.1 imply that $\prec$ is already a topological ordering of G^+, and nothing further needs to be done. Thus, from now on we assume that $k(v) \prec k(u)$ and $V(u) = V(v) = V_{i,j}$ for some $i, j \in [0, |S|]$ in phase III.

Recall the six terminating conditions for the subroutine SEARCH(u, v) used in phase III (see the discussion after Corollary 2.3). We have already assumed that we do not detect any cycle in phase III. Hence, the subroutine SEARCH(u, v) terminates under one of the following four conditions: (C1), (C2), (C5) and (C6). How we update the total order $\prec$ in phase IV depends on the terminating condition under which the subroutine SEARCH(u, v) returned in phase III. In particular, there are two cases to consider.

Case 1. The subroutine SEARCH(u, v) returned under condition (C2) or (C6) in phase III.

In this scenario, we update the total order $\prec$ by calling the subroutine described in Figure 4 (see Section 2.4). In this subroutine, the symbols F_d and B_d respectively denote the set of forward-dead and backward-dead nodes at the end of phase III. Similarly, we will use the symbols F_a and B_a respectively to denote the set of forward-alive and backward-alive nodes at the end of phase III. The subroutine works as follows.

When the subroutine SEARCH(u, v) begins execution in phase III, we had $v \in F_a$ and $u \in B_a$. Since SEARCH(u, v) returned under conditions (C2) or (C6), Property 2.4 implies that $v \in F_d$ and $u \in B_d$ at the end of phase III. Thus, when phase IV begins, let $v, x_1, \ldots, x_f$ be the nodes in F_d in increasing order of priorities, so that $k(v) \prec k(x_1) \prec \cdots \prec k(x_f)$. Similarly, let $y_1, \cdots, y_b, u$ be the nodes in B_d in increasing order of priorities, so that $k(y_1) \prec \cdots \prec k(y_b) \prec k(u)$. By Corollary 2.4, we have $k(x_f) \prec k(y_1)$. Now that the edge (u, v) has been inserted, we need to update the relative ordering among the nodes in $F_d \cup B_d$.

Steps 1-8 in Figure 4 update the total order $\prec$ in such a way that it satisfies the following properties. (1) We still have $k(v) \prec k(x_1) \prec \cdots \prec k(x_f)$. So the relative ordering among the nodes in F_d does not change. (2) Consider any two nodes $x, y \in V$ such that $k(x) \prec k(x_f) \prec k(y)$ at the end of phase III. Then we still have $k(x) \prec k(x_f) \prec k(y)$ at the end of step 8 in Figure 4. So the relative position of x_f among all the nodes in V does not change. (3) The nodes in F_d occur in consecutive positions in the total order $\prec$. Thus, at the end of step 8 it cannot be the case that $k(x') \prec k(x) \prec k(x'')$ if $x \notin F_d$ and $x', x'' \in F_d$.

Claim 2. *Consider any edge (x, y) in $G^-_{i,j}$ where $y \in B_d$ and $x \notin B_d$. Then $k(x) \prec k(v)$ at the end of step 8 in Figure 4.*

Proof. Since $y \in B_d$, $x \notin B_d$ and there is an edge from x to y, Corollary 2.1 implies that $x \in B_a$. Hence, the subroutine SEARCH(u, v) returned under condition (C6), and *not* under condition (C2). By condition (C6), we have $k(x) \prec k(x_f)$ at the end of phase III. Since steps 1-8 in Figure 4 ensure that the nodes in F_d occur in consecutive positions in $\prec$ and they do not change the relative position of x_f among all the nodes in V, we get $k(x) \prec k(v)$ at the end of step 8 in Figure 4. ∎

Steps 9-15 in Figure 4 further update the total order $\prec$ in such a way that it satisfies the following properties. (4) We still have $k(y_1) \prec \cdots \prec k(y_b) \prec k(u)$. In words, the relative ordering among the nodes in B_d does not change. (5) The node u is placed immediately before the node v in the total order $\prec$. This is consistent with the fact that the edge (u, v) has been inserted into the graph G. (6) The nodes in B_d occur in consecutive positions in the total order $\prec$. In other words, at the end of step 15 we cannot find any node $y \notin B_d$ and any two nodes $y', y'' \in F_d$ such that $k(y') \prec k(y) \prec k(y'')$.

To summarize, at this point in time, in the total order $\prec$ the nodes $y_1, \ldots, y_b, u, v, x_1, \ldots, x_f$ occur consecutive to one another, and in this order. Accordingly, Corollary 2.2, Corollary 2.3 and Claim 2 ensure that the total order $\prec$ remains a topological order in G^- at this point in time. Since u appears before v in $\prec$, we also conclude that at this point in time $\prec$ is also a topological order in G^+.

Case 2. The subroutine SEARCH(u, v) returned under condition (C1) or (C5) in phase III.

This case is completely analogous to case 1 above, and we omit its description.

Lemma 2.8. *We spend $\tilde{O}(\sqrt{mn\tau})$ time in phase IV, over the entire sequence of edge-insertions in G.*

Proof. (Sketch) Steps 7 and 14 in Figure 4 can be implemented in $O(1)$ time using the ordered list data structure [8]. Hence, the time spent in phase IV after a given edge-insertion is proportional to the sizes of the sets F_d and B_d at the end of phase III, and by Property 2.5, the sizes of the sets F_d and B_d are (almost) equal to one another. For $l \in [1, m]$, let t_l denote the size of the set F_d at the end of phase III while handling the l^{th} edge-insertion in G. We conclude that the total time spent in phase IV, during the entire sequence of edge-insertions in G, is given by $O\left(\sum_{l=1}^{m} t_l\right)$. The lemma now follows from Corollary 2.5. ∎

2.3 Bounding the Total Update Time of Our Algorithm We simply add up the total time spent by our algorithm in each of these four phases, throughout the entire sequence of edge-insertions in G. In particular, we invoke Lemma 2.2, Lemma 2.3, Lemma 2.7 and Lemma 2.8 and conclude that the total expected update time of our algorithm is at most:

$$\begin{aligned}(2.5) \qquad & \tilde{O}\left(mn/\tau + n^2/\tau + \sqrt{m^3\tau/n} + \sqrt{mn\tau}\right) \\ = \quad & \tilde{O}\left(mn/\tau + \sqrt{m^3\tau/n}\right).\end{aligned}$$

In the above derivation, we have made the assumption that $m = \Omega(n)$. Now, setting $\tau = n/m^{1/3}$, we get a total expected update time of $\tilde{O}(m^{4/3})$. This concludes the proof of Theorem 1.1.

2.4 Pseudocodes for the subroutines in Phase III and Phase IV The relevant pseudocodes for the subroutines in Phase III and Phase IV appear in Figures 1, 2, 3 and 4.

3 Tight Instances For Our Algorithm

We give a brief description of the instance which shows the tightness of our analysis. Before we present the instance, we recall the instance in [10] which shows a lower bound of $\Omega(n\sqrt{n})$ on the total work done by the balanced greedy algorithm. In this instance there are $\sqrt{n}$ directed paths $P_1, P_2, ..., P_{\sqrt{n}}$, each of length $\sqrt{n}$. We will ignore the work done by the balanced greedy algorithm while the adversary constructs $P_1, P_2, ..., P_{\sqrt{n}}$. We assume without loss of generality that all the vertices in P_i precede the vertices in P_{i+1}, for all $i = 1, 2, ...\sqrt{n}$. The adversary's strategy consists of $\sqrt{n} - 1$ rounds. In the first round, the adversary inserts a sequence of $\sqrt{n} - 1$ edges $e_1, e_2, ...e_{\sqrt{n}-1}$ as follows: The first edge e_1 is from the last vertex of path P_2 to the first vertex of path P_1. Upon the insertion e_1, the balanced greedy algorithms moves the entire path P_1 ahead of P_2. The adversary repeats this strategy where the edge e_i is from the last vertex of path P_i to the first vertex of path P_1. Again it is easy to see that at the end of insertion of edge e_i, the path P_1 is between paths P_{i+1} and P_{i+2}. Hence after the insertion of last edge $e_{\sqrt{n}-1}$, the path P_1 moves in front of the all paths. It is easy to see that the total work done by the balanced greedy algorithm during the insertions of edges $e_1, e_2, ...e_{\sqrt{n}-1}$ is $\Omega(\sqrt{n}) \cdot \sqrt{n} = \Omega(n)$, as each edge e_i makes all the vertices of path P_1 to move. This completes the description of the first round. In the remaining rounds, the adversary repeats this strategy on the paths $P_2, P_3, ...P_{\sqrt{n}}$. It is not hard to see that in each of the first $\sqrt{n}/2$ rounds, the balanced greedy algorithm does $\Omega(n)$ amount of work. This implies that the total work done by the balanced greedy algorithm is $\Omega(n\sqrt{n})$.

The instance to show that the analysis of our algorithm is tight is a simple generalization of the above instance. In our instance $m = n$ and $\tau = n^{2/3}$. Let $k = n^{1/3}$. In the beginning, we partition the vertex set into k groups $S_1, S_2, \dots S_k$, each group consisting of exactly n/k vertices. The adversary constructs the lower-bound instance for the balanced greedy on each of the partitions $S_1, S_2, \dots S_k$ separately. Recall that our algorithm partitions the vertex set into n^2/τ bins based on the number of ancestors and descendants a vertex has. Within each bin, our algorithm uses the balanced greedy strategy. Now notice that the maximum number of ancestors or descendants of a vertex in each group S_i is at most $n/k = \tau = n^{2/3}$. Therefore, all the vertices from each group S_i for $i \in [n/\tau]$ fall in the same bin. Hence, the total work done until this stage is $k \cdot \Omega((n/k)\sqrt{n/k}) = \Omega(n^{4/3})$, which is simply the cost of balanced greedy algorithm on the subgraphs induced by the sets $S_1, S_2, \dots S_k$. In the next stage, the adversary repeats the lower bound strategy for the balanced greedy on vertices across groups $S_1, S_2, \dots S_k$. (Think of each group S_i as a meta vertex.) During this phase, each vertex switches exactly n/τ bins. Therefore the total work done by our algorithm in moving vertices across the bins is $\Omega(n/\tau) \cdot n = \Omega(n^{4/3})$. Thus, we observe that the total work done by our algorithm in moving the vertices across the bins is equal to the total work done within the bins, which is the quantity our algorithm balances. As $m = n$ in our instance, we complete the proof.

References

[1] A. Abboud and V. V. Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *FOCS*, 2014.

[2] D. Ajwani and T. Friedrich. Average-case analysis of incremental topological ordering. *Discrete Applied Mathematics*, 158(4):240–250, 2010.

[3] D. Ajwani, T. Friedrich, and U. Meyer. An $O(n^{2.75})$ algorithm for incremental topological ordering. *ACM Trans. Algorithms*, 4(4):39:1–39:14, 2008.

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

[5] M. A. Bender, J. T. Fineman, S. Gilbert, and R. E. Tarjan. A new approach to incremental cycle detection and related problems. *ACM Trans. Algorithms*, 12(2):14:1–14:22, 2016.

[6] A. Bernstein and S. Chechik. Incremental topological sort and cycle detection in expected $\tilde{O}(m\sqrt{n})$ total time. In *SODA*, 2018.

[7] E. Cohen, A. Fiat, H. Kaplan, and L. Roditty. A labeling approach to incremental cycle detection. *CoRR*, abs/1310.8381, 2013.

```
01. INITIALIZE: $F_a = \{v\}$, $B_a = \{u\}$, $F_d = \emptyset$ and $B_d = \emptyset$.
02. WHILE $F_a \neq \emptyset$ AND $B_a \neq \emptyset$:
03.     $x = \arg\min_{x' \in F_a}\{k(x)\}$.
04.     IF $k(x) \succ \min_{y' \in B_d}\{k(y')\}$, THEN
05.         RETURN NO.      // Insertion of the edge $(u, v)$ does not create a cycle.
06.     ELSE
07.         EXPLORE-FORWARD($x$).
09.     $y = \arg\max_{y' \in B_a}\{k(y')\}$.
10.     IF $k(y) \succ \max_{x' \in B_d}\{k(x')\}$, THEN
11.         RETURN NO.      // Insertion of the edge $(u, v)$ does not create a cycle.
12.     ELSE
13.         EXPLORE-BACKWARD($y$).
14. RETURN NO.              // Insertion of the edge $(u, v)$ does not create a cycle.
```

Figure 1: Subroutine: SEARCH(u, v) used in phase III.

```
1. $F_a = F_a \setminus \{x\}$ and $F_d = F_d \cup \{x\}$.
2. FOR ALL $(x, x') \in E$ with $V(x) = V(x')$:
3.     IF $x' \in B_a \cup B_d$, THEN
4.         RETRUN YES.      // Insertion of the edge $(u, v)$ creates a cycle.
5.     ELSE IF $x' \notin F_a \cup F_d$, THEN
6.         $F_a = F_a \cup \{x'\}$.
```

Figure 2: Subroutine: EXPLORE-FORWARD(x) used in phase III.

```
1. $B_a = B_a \setminus \{y\}$ and $B_d = B_d \cup \{y\}$.
2. FOR ALL $(y', y) \in E$ with $V(y') = V(y)$:
3.     IF $y' \in F_a \cup F_d$, THEN
4.         RETRUN YES.      // Insertion of the edge $(u, v)$ creates a cycle.
5.     ELSE IF $y' \notin B_a \cup B_d$, THEN
6.         $B_a = B_a \cup \{y'\}$.
```

Figure 3: Subroutine: EXPLORE-BACKWARD(y) used in phase III.

```
01. $Q = F_d$.
02. $x^* = \arg\max_{x \in Q}\{k(x)\}$
03. $Q = Q \setminus \{x^*\}$.
04. WHILE $Q \neq \emptyset$:
05.     $x' = \arg\max_{x \in Q}\{k(x)\}$.
06.     $Q = Q \setminus \{x'\}$.
07.     INSERT-BEFORE($x', x^*$).
08.     $x^* = x'$.
09. $y^* = v$.
10. $Q = B_d$.
11. WHILE $Q \neq \emptyset$:
12.     $y' = \arg\max_{y \in Q}\{k(y)\}$.
13.     $Q = Q \setminus \{y'\}$.
14.     INSERT-BEFORE($y', y^*$).
15.     $y^* = y'$.
```

Figure 4: Subroutine: UPDATE-FORWARD(.) used in phase IV.

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

[9] B. Haeupler, T. Kavitha, R. Mathew, S. Sen, and R. E. Tarjan. Faster algorithms for incremental topological ordering. In *ICALP (Tack A)*, 2008.

[10] B. Haeupler, T. Kavitha, R. Mathew, S. Sen, and R. E. Tarjan. Incremental cycle detection, topological ordering, and strong component maintenance. *ACM Trans. Algorithms*, 8(1):3:1–3:33, 2012.

[11] M. Henzinger, S. Krinninger, D. Nanongkai, and T. Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *STOC*, 2015.

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

[13] T. Kopelowitz, S. Pettie, and E. Porat. Higher lower bounds from the 3sum conjecture. In *SODA*, 2016.

[14] H. Liu and K. Chao. A tight analysis of the Katriel-Bodlaender algorithm for online topological ordering. *Theor. Comput. Sci.*, 389(1-2):182–189, 2007.

[15] A. Marchetti-Spaccamela, U. Nanni, and H. Rohnert. Maintaining a topological order under edge insertions. *Inf. Process. Lett.*, 59(1):53–58, 1996.

[16] D. J. Pearce and P. H. J. Kelly. A dynamic topological sort algorithm for directed acyclic graphs. *ACM Journal of Experimental Algorithmics*, 11, 2006.