A New Approach to Incremental Topological Ordering

Michael A. Bender* Jeremy T. Fineman† Seth Gilbert‡

Abstract

Let $G = (V,E)$ be a directed acyclic graph (dag) with $n = |V|$ and $m = |E|$. We say that a total ordering $\prec$ on vertices V is a ***topological ordering*** if for every edge $(u,v) \in E$, we have $u \prec v$. In this paper, we consider the problem of maintaining a topological ordering subject to dynamic changes to the underlying graph. That is, we begin with an empty graph $G = (V,\emptyset)$ consisting of n nodes. The adversary adds m edges to the graph G, one edge at a time. Throughout this process, we maintain an *online* topological ordering of the graph G.

In this paper, we present a new algorithm that has a total cost of $O(n^2 \log n)$ for maintaining the topological ordering throughout all the edge additions. At the heart of our algorithm is a new approach for maintaining the ordering. Instead of attempting to place the nodes in an ordered list, we assign each node a label that is consistent with the ordering, and yet can be updated efficiently as edges are inserted. When the graph is dense, our algorithm is more efficient than existing algorithms. By way of contrast, the best known prior algorithms achieve only $O(\min(m^{1.5}, n^{2.5}))$ cost.

1 Introduction

Let $G = (V,E)$ be a directed acyclic graph (dag) with $n = |V|$ and $m = |E|$. We say that a total ordering $\prec$ on vertices V is a ***topological ordering*** if for every edge $(u,v) \in E$, we have $u \prec v$. Given a specific dag G, there are two well-known approaches for finding a topological ordering in $O(n+m)$, either by depth-first search, or by repeated deletion of vertices with no incoming edges.

This paper addresses an incremental variant of this problem, which arises in a variety of contexts, including compilers [14, 16], deadlock detection [4], pointer analysis [17, 18], and incremental circuit evaluation [3].

In the problem of incremental topological ordering, our goal is to maintain a topological ordering even as edges are added to the graph. Initially, the graph G is unknown; edges are added one at a time. After each edge addition, we must recalculate a valid topological ordering. More specifically, the goal is to maintain a data structure that supports two operations: (1) edge insertions, in which a new edge is added to the graph G; and (2) queries of the form: "Does u come before v in the topological ordering?"[1] In this paper, as well as in previous work, queries are answered in $O(1)$ time; the key question is how fast can edge insertions be processed?

Prior Work. The simplest solution to the problem of incremental topological ordering is to recalculate a new ordering after each edge insertion, resulting in $O(n(m+n))$ time for m edge insertions. In recent years, there have been several significant improvements. Marchetti-Spaccamela et al. [15] gave the first nontrivial solution, handling m insertions in a total of $O(mn)$ time. Alpern et al. [3] gave an algorithm that performs well in a greedy sense: given a topological ordering and an edge insertion, their algorithm performs (almost) the minimum amount of work possible to find a new topological ordering. (This form of analysis, however, says little about the total time to perform m edge insertions.) Katriel and Bodlaender [11] gave a variant of the algorithm introduced by Alpern et al., which they show to run in $O(\min\{m^{3/2}\log n, m^{3/2} + n^2 \log n\})$ time. They also showed significantly better bounds for graphs of bounded treewidth. Liu and Chao [13] gave a tighter analysis of the Katriel-Bodlaender algorithm, showing that it runs in $O(m^{3/2} + mn^{1/2}\log n)$ time. Kavitha and Mathew [12] gave a slightly better variant taking $O(m^{3/2} + m^{1/2} n \log n)$. Most recently, Haeupler et al. [8, 9] gave a variant of the Alpern et al. / Katriel-Bodlaender algorithm that runs in time $O(m^{3/2})$.

The algorithms mentioned thus far do no better than $O(n^3)$ for dense graphs where $m = \Theta(n^2)$. Ajwani et al. [2] gave the first improvement for dense graphs, exhibiting an algorithm that runs in $O(n^{2.75})$ time for any number of edge insertions. Haeupler et al. [8, 12] improved this algorithm, resulting in a simpler algorithm requiring only $O(n^{2.5})$ time. This bound is not known to be tight, however, so this algorithm's true running time could potentially match our own.

Pearce and Kelly [18] gave an algorithm that they showed to be fast in practice on random sparse graphs, but

*Dept. of Computer Science, Stony Brook University, bender@cs.sunysb.edu

†CSAIL, MIT, jfineman@csail.mit.edu

‡EPFL, seth.gilbert@epfl.ch

§This research was supported in part by NSF grants CCF-0621511, CNS-0615215, CCF-0541209, CCF-0621439/0621425, CCF-0540897/05414009, CCF-0634793/0632838, and CNS-0627645.

[1] Although there may be many valid topological orderings, all answers given by the data structure after the kth edge insertion/deletion must be consistent with *the same* topological ordering.

that is provably worse than that Alpern et al. algorithm in the worst case. Ajwani and Friedrich [1] proved that the Alpern et al., Katriel and Bodlaender, and Pearce and Kelly algorithms all take expected time $O(n^2 \mathrm{polylog}(n))$ for edges forming a complete graph inserted in a random order.

The only nontrivial general lower bound that we are aware of is an $\Omega(n \log n)$ lower bound for $n-1$ edge insertions due to Ramalingam and Reps [19]. Katriel [10] gives an $\Omega(n^2)$ lower bound when $m = O(n)$ for algorithms that explicitly maintain the rank of each vertex in the topological order. Of known algorithms, this lower bound only applies to those algorithms that store the topological ordering in an array—those of Marchetti-Spaccamela et al., Pearce and Kelly, and Ajwani et al. It does not apply to our algorithm, nor does it apply to any of the sparse-graph algorithms. Haeupler et al. [9] give an $\Omega(nm^{1/2})$ lower bound for algorithms that only update the "affected region" of the topological ordering on an edge insertion. This lower bound applies to all previous algorithms, but it does not apply to our algorithm.

Our Results. In this paper, we present a new algorithm that takes $O(n^2 \log n)$ time to support any number of edge insertions. Our analysis is tight in that there exist graphs and edge-insertion sequences causing our algorithm to run in $\Theta(n^2 \log n)$ time. This bound beats the $O(n^{2.5})$ bound of Haeupler et al. [8, 12] and beats the $O(m^{3/2})$ bound of Haeupler et al. [8,9] whenever $m \geq n^{4/3} \log^{2/3} n$. We analyze our algorithm in the RAM model (as is the case for all prior algorithms).

Our approach is quite different from previous algorithms. Typically, a topological ordering is maintained explicitly as either a linked list or an array. When adding an edge (u, v), the algorithm first checks whether u appears before or after v in the existing topological ordering; if u appears *after* v, then the array or linked list is updated so that u precedes v in the ordering, as is required by the insertion of edge (u, v). During the insertion, the algorithm modifies only vertices in the "affected region" of the list/array, i.e., those vertices that lie between v and u. The key to these algorithms is to efficiently discover which vertices in the affected region need to be moved.

Our algorithm, by contrast, does not maintain an array or linked list, but instead assigns a *label* to each vertex in the graph, reassigning labels as edges are inserted.[2] The labels induce a topological ordering on the dag and are also used to assist in efficient updates during edge insertions.

Our data structure can be readily extended in various ways. First, it can be augmented to support additional operations, such as queries of the form: "What is u's successor (or predecessor) in the topological ordering?" (Such queries are supported by most previous algorithms.) Second, it can be augmented to detect cycles in the graph G; we assume throughout this paper that the graph G is acyclic; however, with a small amount of bookkeeping we can detect anomalous graphs. Finally, our data structure can readily support edge deletions as well as edge insertions; however, performance guarantees apply only to executions consisting only of insertions. We comment on these extensions in Section 3.

Roadmap. The remainder of the paper is organized as follows. First, in Section 2, we present some preliminary definitions, along with an overview of our approach for maintaining a topological ordering. As a simple example of this approach, we give an algorithm that achieves $O(mn)$ running time. Next, in Section 3, we present our new algorithm in detail. Finally, we analyze the algorithm in Section 4, and conclude in Section 5.

2 Basic strategy

This section describes our basic strategy for maintaining a topological ordering. We show how this strategy can be applied in a simple fashion to achieve an $O(mn)$ algorithm for n vertices and m edge insertions. While this bound is not new, it demonstrates a quite different approach. In Section 3 we show how this same basic approach can be more carefully applied to develop a more efficient algorithm that yields running time $O(n^2 \log n)$.

Terminology. Given a dag $G = (V, E)$, we say that a node u is a ***predecessor*** of a node v (and that v is a ***successor*** of u), if there is a directed path from u to v in G. Notice that, according to this terminology, a node u is a predecessor (and successor) of itself. When the edge (u, v) exists, we say that u is an ***immediate predecessor*** of v (and v is an immediate successor of u).

Labels and Orders. We associate with each vertex u an (integer) label $L(u)$. From these labels, we derive a total order in the natural way: if $L(u) < L(v)$, then $u \prec v$. If $L(u) = L(v)$, then we break the tie in an arbitrary but consistent fashion, say, using the unique identifiers of the vertices in question.

It is easy to see that the ordering induced by the labels is a topological ordering as long as the labels are consistent with the underlying dag. That is, whenever (u, v) is an edge in E, we have $L(u) < L(v)$.

Labels are updated dynamically as edges are added to the graph. When adding edge (u, v), we update the label at v, if necessary, along with some subset of v's successors. The first step involves determining whether the label of v needs to be increased. We refer to this process as ***visiting*** v. If v's label needs to be increased, a new value of the

[2] Previous linked-list-based algorithms implicitly use labels to perform queries as part of an order-maintenance data structure [5, 7]. By contrast, our labels have semantic meaning within the graph itself and play a key role in determining which vertices to update.

label is chosen. After visiting a node v, we perform a truncated depth-first search on the dag, starting at v and visiting successors of v in turn. When we traverse an edge in the dag, we say that we are ***following*** the edge. Choosing when to follow edges and how to update labels is at the heart of achieving an efficient algorithm.

A simple algorithm. As an example of how to apply this paradigm, we consider a simple algorithm in which the label $L(v)$ represents the ***depth*** of v, where depth is defined as usual: if v has no immediate predecessors, then $L(v) = 0$; otherwise, $L(v) = \max_{u:(u,v)\in E} L(u) + 1$. If each node is labelled with its depth, it is clear that the labelling induces a topological ordering: if u precedes v in the dag G and $u \neq v$, then $L(u) < L(v)$.

Initially, as there are no edges in the graph G, $L(u) = 0$ for every vertex u. Every time an edge is added to the dag, the labels on the vertices are updated to reflect the changes in depth caused by the new edge. Specifically, when adding edge (u,v), we compare the labels at u and v: if $L(u) < L(v)$, then the depth of v remains unchanged. If, however, $L(u) \geq L(v)$, then we update v's label: $L(v) \leftarrow L(u)+1$. We then recursively follow all of v's outgoing edges, recursively performing the same "update label/follow edges" procedure at each of v's immediate successors. When this depth-first traversal terminates, each vertex is labelled with its depth in the graph.

There is a straightforward analysis of this algorithm. For each vertex u, the maximum depth is $n-1$, and hence $L(u) \leq n-1$. Since labels/depth are nondecreasing, it follows that $L(u)$ increases at most $n-1$ times, and thus the total cost of updating labels is $O(n^2)$. The remaining cost comes from following edges. (Recall that we may sometimes follow an edge (v,w) but not update the label at w, as it is already sufficiently large.) Notice that the algorithm follows the edge (v,w) only when (1) (v,w) is added, or (2) v's label increases. Since the label at v increases at most $n-1$ times, each edge can be followed at most n times. We conclude that the total cost to insert m edges is $O(mn)$.

3 Algorithm for Dense Graphs

This section describes an $O(n^2 \log n)$ algorithm for incremental topological sort, which is based on the basic strategy described in Section 2. The analysis appears in Section 4.

The example in Figure 1 demonstrates a shortcoming of the simple $O(mn)$ algorithm. Consider a bipartite clique with vertices $v_1, v_2, \ldots, v_k$ and $w_1, w_2, \ldots, w_k$, and an additional source vertex u that has a directed edge to every v_i. Consider an execution of the simple algorithm. When the depth of u increases, the algorithm follows each outgoing edge (u, v_i), and the label of each v_i also increases. Since v_i's label increases, the algorithm also follows each edge (v_i, w_j). Thus, whenever the depth of u increases, the algorithm

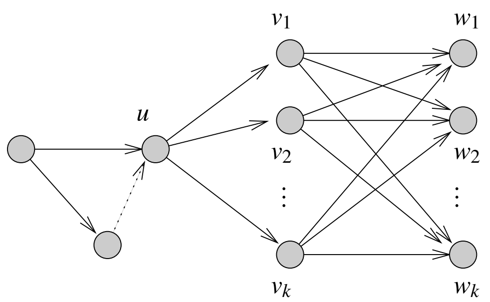

Figure 1: When the dotted edge is added, the depth of u increases by 1. The simple algorithm performs $\Theta(k^2)$ work, following edges (v_i, w_j) for all $1 \leq i, j \leq k$.

follows *all of the* $\Theta(k^2)$ edges in the clique. Setting $k = \Theta(n)$ and increasing the depth of u by one $\Theta(n)$ times yields a total cost of $\Theta(n^3)$.

The main goal of our algorithm is to reduce the number of times that we visit a particular vertex w, that is, to bound the number of visits by $O(n \log n)$. (By contrast, the simple algorithm may visit each node up to $\Omega(n^2)$ times.)

Key Modifications. In order to achieve better performance, we make two key modifications to the simple algorithm. First, we do not blindly follow all outgoing edges from v whenever v's label increases. Instead, for each edge (v,w), we cache the value of w's label at v and only follow the edge (v,w) if the cached information indicates that w's label needs to increase. Specifically, we associated with each edge (v,w) the value $cache(v,w)$, which records the label of w as of the last time the algorithm followed edge (v,w). Since labels are nondecreasing, $cache(v,w) \leq L(w)$. Thus, we do not follow (v,w) unless $L(v) \geq cache(v,w)$.

This first improvement alone does not improve the worst-case running time of our algorithm, as exhibited again by the bipartite-clique example in Figure 1: after each edge insertion completes, the cache at each edge leaving v_i correctly records the depth of each node w_j; however, when v_i increases by 1 on the next edge insertion, we must again follow all outgoing edges from v_i.

Our second modification is to use a more aggressive label-update rule. If we could update the label of a node by a larger quantity (instead of incrementing it), then we could cache that larger value and avoid unnecessary visits. However, we want to avoid increasing the label too much, which may result in labels growing too big.

Thus, we no longer constrain the label of a node to represent its depth, as it is expensive to maintain a node's precise depth. Rather, we consider the label $L(v)$ to approximate the total number of predecessors (not just immediate) of v. (Notice that the depth of a node is a very loose lower bound on

State maintained by each node $u \in V$:

$L(u)$, the label of u.
$\forall j \in \{0,1,\ldots,\lg n\} : N(u,j)$, a counter bounded by 2^{j+2}.
$\forall j \in \{0,1,\ldots,\lg n\} : \Lambda(u,j)$, an old label of u.
$\forall v : (u,v) \in E$: *cache*(u,v), an old label of v.
outgoing(u), an array of outgoing edges.

FOLLOW(u,v)

1 **if** $L(u) \geq L(v)$
2 **then** $L(v) \leftarrow L(u)+1$
3 **else** $\triangleright$ Check if v has sufficiently many immediate
$\triangleright$ predecessors to increase its label.
4 $j \leftarrow \lceil \lg(L(v)-L(u)) \rceil$
5 $N(v,j) \leftarrow N(v,j)+1$
6 **if** $N(v,j) = 2^{j+2}$
7 **then** $L(v) \leftarrow \max(L(v), \Lambda(v,j)+2^j)$
8 $N(v,j) \leftarrow 0$
9 $\Lambda(v,j) \leftarrow L(v)$
10 **if** $L(v)$ has increased:
11 **then for each** (v,w) such that *cache*$(v,w) \leq L(v)$
12 **do** FOLLOW(v,w)
$\triangleright$ Done following the edge.
$\triangleright$ Update (v,w) in u's outgoing edge set.
13 *cache*$(u,v) \leftarrow L(v)$
14 Update (u,v) in *outgoing*(u).

Figure 2: Pseudocode for updating labels, when following an edge in the dag. The triangles denote comments. Each node v maintains four data structures: a label $L(v)$, a counter $N(v,j)$ and old label $\Lambda(v,j)$ for $0 \leq j < \lg(n)$, and an outgoing-edge set *outgoing*(v) organized by *cache*(v,w) — a cached version of w's label.

the total number of predecessors.) That is, we assign a label $L(v)$ such that $0 \leq L(v) \leq Pred(v) < n$, where $Pred(v)$ is the total number of predecessors of v. (Of course, the labels also must induce a valid topological order.)

When increasing v's label, the magnitude of the increase depends on the number of immediate predecessors known to have large labels: the more predecessors of v that have large labels, the higher we increment the label of v. This strategy captures the intuition that if a vertex has many immediate predecessors with large labels, then it most likely has a very large number of predecessors (immediate or otherwise) in the dag. By increasing the label significantly, we prevent too frequent updates as further edges are added.

The Algorithm. As previously noted, our algorithm maintains the value $L(v)$ (initially 0), with $0 \leq L(v) < n$. We associate with each edge (u,v) the value *cache*(u,v), which stores the (old) value of $L(v)$ when the algorithm last followed edge (u,v). (Note that the cached value may be out-of-date, as $L(v)$ may have increased since then). We organize all outgoing edges into a data structure *outgoing*(v), which we describe later.

We also associate with v a collection of counters $N(v,j)$ and corresponding labels $\Lambda(v,j)$, for each $j : 0 \leq j < \lg(n)$, all initially 0. The jth counter $N(v,j)$ satisfies the following invariant: $0 \leq N(v,j) \leq 2^{j+2}$. The label $\Lambda(v,j)$ stores the value of v's label when $N(v,j)$ was last reset to 0. The $N(v,j)$ counter counts the number of incoming edges (u,v) that, when last followed, had $2^{j-1} < L(v)-L(u) \leq 2^j$.

We now describe the algorithm in more detail. When inserting the edge (u,v) into the dag, we proceed as follows: (1) initialize *cache*$(u,v) \leftarrow 0$; (2) insert (u,v) into *outgoing*(u);, and (3) follow the edge (u,v) as described by FOLLOW(u,v) in Figure 2. This procedure updates v's label and other state, and then (selectively) calls FOLLOW$(v,*)$ recursively on the ongoing edges of v.

When executing FOLLOW(u,v), we first check whether $L(u) < L(v)$. If not, we increase $L(v)$ by 1 in line 2, thus ensuring that the resulting labelling induces a valid topological ordering.

If, on the other hand, we already have $L(u) < L(v)$, then we examine whether v should have its label increased due to its counts of immediate predecessors. This step is where the label is incremented more aggressively than in Section 2.

First, we find the magnitude of difference between $L(u)$ and $L(v)$ in line 4—specifically, we find the smallest j such that $L(v)-L(u) \leq 2^j$. We then increment the corresponding counter $N(v,j)$. If the counter $N(v,j)$ reaches its maximum value of 2^{j+2}, then we perform the our "more aggressive" update of v's label, increasing $L(v)$ by up to 2^j-1 in line 7. We next reset $N(v,j)$ to 0 and record the corresponding $\Lambda(v,j)$ in lines 8 and 9.

This label update may seem strange at this point, but we will show in Lemma 4.1 that when $\Lambda(v,j)+2^j > L(v)$, v has at least 2^{j+1} distinct immediate predecessors with label at least $\Lambda(v,j)-2^j$. Thus, we know that v has at least $\Lambda(v,j)-2^j+2^{j+1}$ predecessors, and we update the label accordingly.

At this point, if v's label increases (whether in line 2 or line 7), then we follow all outgoing edges whose targets have cached labels that are $\leq L(v)$, i.e., targets that may need to have their label increased. Finally, when the algorithm has finished following outgoing edges, we associate v's new label $L(v)$ with *cache*(u,v) and update the edge (u,v) in u's outgoing edge list.

We next comment on some of the implementation details.

Implementing the Outgoing-Edge Set. The outgoing-edge set data structure is straightforward to implement. The variable $outgoing(v)$ is an n-slot array, each slot containing a linked list of outgoing edges. If $cache(v,w) = x$, then a pointer to w is stored in a linked list at slot x in the array. Notice that updating an edge (i.e., in line 14) is easy in that it simply involves removing it from one linked list and adding it to another.

To determine which edges to follow (in line 11), we follow all outgoing edges in array slots between v's old label and v's new label. After following each edge, the target's label (and hence the edge's cached label) has increased beyond v's current label $L(v)$. Thus, when $\text{FOLLOW}(u,v)$ returns, there are no outgoing edges stored in array slots $0,\ldots,L(v)$. As a result, we need only examine each slot of the array once during an execution.

Calculating in the RAM model. Some machine models allow for a constant-time lg calculation, but we do not require such an assumption. Since we only compute this logarithm for at most n different values (i.e., the difference in the possible labels), we can pre-compute a size-n logarithm table with all the necessary entries. Even a naive algorithm for computing base-2 logarithms of $\log(n)$-bit numbers requires only $O(\log(n))$ time using only additions (repeated doubling) and comparisons. The $O(n\log(n))$ time for pre-computing the entire table is dwarfed by the $O(n^2\log(n))$ time of the topological-ordering algorithm.

Similarly, we could pre-compute a table for calculating 2^{j+2} using only additions, thereby not requiring a bitshift operation in the machine model.

Supporting Predecessor/Successor queries. As described so far, our data structure supports only queries of the form, "Does u precede v in the topological ordering?" It does not (efficiently) support queries of the form, "What is the next vertex in the topological ordering after u?" These queries are easy to support without increasing the asymptotic running time.

Throughout the execution of the algorithm, maintain a linked list matching the topological ordering. To search into the linked list, also maintain a balanced search tree (BST) (see [6], Chapter 13), ordered/keyed by label, where ties are broken by unique vertex identifiers. Initially the vertices are simply sorted by identifier. Whenever v's label increases, remove v from the tree and reinsert it with key equal to its new label. Query the BST for v's predecessor u, and move v from its current location in the linked list, instead locating v after its BST-predecessor u. Since labels have maximum value of $n-1$ (shown later in Lemma 4.1), a node v is reinserted at most $n-1$ times, resulting in n BST insertions and predecessor queries, as well as n linked-list moves, per vertex. Each BST operations has cost $O(\log n)$ (using a reasonable BST implementation), and each linked-list operation has constant cost. The total additive cost of $O(n^2 \log n)$ for the $O(n^2)$ BST operations has no effect on the asymptotic running time of our algorithm.

Detecting a Cycle. Thus far, we have assumed that no edge insertion introduces a cycle to the graph. It is easy to modify our algorithm to detect the introduction of a cycle, although the performance bounds only hold until the first cycle is detected; that is, we provide no mechanism for deleting an edge aside from rebuilding the entire data structure.

To detect whether the addition of (u,v) introduces a cycle, simply check whether u is ever visited during the depth-first search performed while following edges. If so, there is a cycle. The nodes in the cycle can be found by a separate graph search.

4 The Analysis

We now analyze the algorithm from Section 3. There are three key theorems. First, we argue that the ordering induced by the labels is, in fact, a topological order. Second, we argue that labels are bounded by n. Lastly, we analyze the cost of edge insertions, demonstrating that the total cost of inserting m edges is $O(n^2 \log n)$, and we show this analysis to be tight. The key observation is that no edge is followed too many times.

Correctness of the Topological Ordering. We begin by showing that the data structure maintains a valid topological ordering:

THEOREM 4.1. *After completely processing each edge insertion, if $((u,v)) \in E$, then $L(u) < L(v)$.*

Proof. Initially, the labels trivially induce a good ordering since there are no edges in the dag. Assume, for the sake of contradiction, that after some edge insertion the theorem is violated. Consider the first edge insertion causing a violation, and let edge (u,v) be an edge for which $L(u) > L(v)$. During the edge insertion, the label $L(u)$ must have increased, thereby causing the violation. If we subsequently followed (u,v), then $L(v)$ would have increased beyond $L(u)$ (in line 2).

Suppose, therefore, that we did not follow (u,v) after the last increase to $L(u)$. It follows that $cache(u,v) > L(u)$, as the edge was not followed in lines 11–12. We know that the cached label $cache(u,v) \le L(v)$, since $L(v)$ is nondecreasing. Hence, $L(v) \ge cache(u,v) > L(u)$, which is a contradiction. □

Bounded Labels. We next show that the value of each label is bounded by n. (Notice that this fact also ensures that each label can be stored in a single word, which has been implicit throughout.)

LEMMA 4.1. *For $v \in V$, let $Pred(v)$ be the total number of predecessors of v. Then at any point during the algorithm execution, $L(v) \leq Pred(v) < n$.*

Proof. We proceed by induction over the number of edges that have been followed. Initially, $L(v) = 0$, and the claim holds trivially. Assume the lemma is true before following the edge (u,v). We show that it holds after following the edge. If label $L(v)$ does not increase, then the claim follows immediately, as $L(v) \leq Pred(v)$ by inductive hypothesis.

Suppose instead that $L(v)$ increases while following (u,v). There are two places that v's label increases: line 2 and line 7 of Figure 2. In the first case, $L(v) \leftarrow L(u)+1$. By inductive hypothesis, $L(u) \leq Pred(u)$, and we know that $Pred(u) < Pred(v)$ since the predecessors of v include all the predecessors of u plus the node v itself. (Recall that v is considered to be a predecessor of itself.) Thus, $L(v) = L(u)+1 \leq Pred(u)+1 \leq Pred(v)$, and the claim follows.

In the second case, $L(v)$ increases in line 7 as a result of $N(v,j)$ increasing to 2^{j+2}. We fix j for the remainder of this proof. First, we observe that $L(v)$ increasing here implies that (prior to the update) $L(v) < \Lambda(v,j)+2^j$. (Otherwise the label $L(v)$ would remain unchanged.) Consider the 2^{j+2} increases to $N(v,j)$ since the last time it was reset to 0.

We claim that each distinct edge contributed at most 2 such increases to $N(v,j)$. Suppose for the sake of contradiction that some edge (u,v) contributed 3 or more increases to $N(v,j)$, and consider the last 3 such increases. For $x \in \{u,v\}$, let $L_1(x)$, $L_2(x)$, and $L_3(x)$ be the values of x's label immediately prior to these 3 increases, respectively. Since the counter increases, $L_i(v) - L_i(u) > 2^{j-1}$. (Recall that this inequality follows from the choice of j, see line 4.)

Moreover, since $cache(u,v)$ is updated after each edge is followed, and since the edge is only followed if the label $L(u)$ exceeds the cached value of $L(v)$, we can conclude that $L_2(u) \geq L_1(v)$ and $L_3(u) \geq L_2(v)$. Combining these two facts, we conclude that $L_2(v) > L_2(u)+2^{j-1} \geq L_1(v)+2^{j-1}$ and $L_3(v) > L_3(u)+2^{j-1} \geq L_2(v)+2^{j-1}$, which together imply that $L_3(v) > L_1(v)+2^j$. Finally, since the counter does not reset between these increases, and $\Lambda(v,j)$ represents v's label at the time of the previous reset, $L_1(v) \geq \Lambda(v,j)$. It follows that $L(v) \geq L_3(v) > \Lambda(v,j)+2^j$, which is a contradiction.

We therefore conclude that each edge contributes at most 2 increases to $N(v,j)$. Since there have been 2^{j+2} increases when the counter resets (causing the label of v to be increased), we conclude that there are at least $2^{j+2}/2$ distinct edges contributing to $N(v,j)$. Fix some u such that (u,v) contributes to $N(v,j)$. Each time edge (u,v) contributes to the count, we have $L(u) \geq L(v) - 2^j$ (again, by the way in which j was chosen on line 4). Thus, since $L(v) \geq \Lambda(v,j)$, we know that $L(u) \geq \Lambda(v,j) - 2^j$.

Thus, v has at least 2^{j+1} distinct immediate predecessors with label at least $\Lambda(v,j) - 2^j$. We call these vertices the "contributing predecessors."

Let x be a topologically earliest contributing predecessor. By inductive hypothesis, we have $Pred(x) \geq L(x)$. All of x's predecessors are predecessors of v. Moreover, none of the other contributing predecessors, nor v itself, are predecessors of x. Thus, $Preds(v) \geq Preds(x)+2^{j+1} \geq L(x)+2^{j+1} \geq (\Lambda(v,j)-2^j)+2^{j+1} = \Lambda(v,j)+2^j$. Noting that the label $L(v)$ is increased to $\Lambda(v,j)+2^j$ completes the proof. □

Complexity Analysis of Insertions. We now show that the total cost of up to m edge insertions is $O(n^2 \log n)$. The key observation is that no vertex is visited too many times, specifically, more than $O(n \log n)$ times. The main idea of the proof is to amortize the cost of following an edge directed towards v against the number of times that the label of v is increased.

LEMMA 4.2. *In every execution, for every $v \in V$, vertex v is visited at most $O(n \log n)$ times.*

Proof. Whenever we visit v, we either increases $L(v)$ (line 2), or we increase $N(v,j)$ for some j (line 5). The former occurs at most $n-1$ times over the course of the algorithm.

For the latter case, consider the jth counter. Whenever the counter reaches 2^{j+2} it resets. Let $\Lambda_i(v,j)$ denote the value of v's label associated with the ith reset. Observe $\Lambda_{i+1}(v,j) \geq \Lambda_i(v,j)+2^j$ due to the update in line 7, or more generally $\Lambda_i(v,j) \geq i2^j$. Since $n > L(v) \geq \Lambda(v,j)$, it follows that the maximum value of i here is $\lfloor (n-1)/2^j \rfloor$, and hence $N(v,j)$ can be increased at most

$$2^{j+2}\left(\lfloor (n-1)/2^j \rfloor + 1\right) \leq \frac{2^{j+2}n}{2^j} + 2^{j+2} \leq 4n + 2^{j+2}$$

times. Summing over all $O(\log n)$ values of j gives $O(n \log n)$ counter increments and hence visitations of v. □

Combining Lemma 4.2 with a cost analysis of each call to FOLLOW yields our total running time:

THEOREM 4.2. *The total running time to perform up to m edge insertions is $O(n^2 \log n)$.*

Proof. To reach this bound, we calculate the cost of each call to FOLLOW, and multiply by the total number of times that any node is visited. Ignoring lines 11–12, each step of FOLLOW has constant cost. The only remaining cost is due to finding edges to follow in the outgoing-edge data structure. The outgoing-edge data structure is a size-n array of linked lists. We visit each array cell only once over the entire course of the algorithm, for an aggregate cost of $O(n)$. We charge the $O(1)$ cost of traversing the linked list against the outgoing edge followed. Multiplying the array-traversal cost for each outgoing-edge data structure by n

vertices yields a total cost for our algorithm of $O(n^2+F)$, where F is the total number of edge followings. Applying Lemma 4.2 completes the proof. □

Finally, we show that our analysis is tight.

THEOREM 4.3. *For any sufficiently large n, there exists a sequence of $\Theta(n^2)$ edge insertions on an n-vertex dag that causes our algorithm to follow $\Omega(n^2 \log n)$ edges.*

Proof. Without loss of generality, suppose $n = 3k - 4$, where $k \geq 2^3$ is a power of 2. The graph we construct consists of three categories of vertices: (1) vertices $u_0, u_1, \ldots, u_{k-1}$, (2) sets of vertices $S_0, S_1, \ldots, S_{\lg(k)-3}$ with $|S_j| = 2^{j+2}$ (so $\sum_j |S_j| = k-4$), and (3) a set of vertices T with $|T| = k$. Initially there are no edges in the graph, and all labels are 0.

First, add edges (u_i, u_{i+1}) in order for $0 \leq i < k-1$. After these edge additions, $L(u_i) = i$. These labels are invariant over the remainder of the edge insertions — we use these vertices as anchors to increase the labels of all the other vertices. In fact, the *only* time the labels of any other vertex $v \in (\bigcup_j S_j) \cup T$ will increase is when adding an edge (u_i, v).

The edge insertions proceed in phases ranging from 1 to k. In phase i, first insert edge (u_{i-1}, t) for all $t \in T$, thereby increasing $L(t)$ such that $L(t) = i$. Next, consider each j for which i is a multiple of 2^j. There are two cases.

Case 1: If $i = 2^j$, add edges (s_j, t) for all $s_j \in S_j$ and $t \in T$. Observe that before the edge addition, $N(t, j) = 0$, $\Lambda(t, j) = 0$, and $L(s_j) = 0 = L(t) - 2^j$. After the 2^{j+2}th edge insertion, $N(t, j)$ reaches 2^{j+2}. We have, however, that $L(t) \geq \Lambda(t, j) + 2^j$, and hence $L(t)$ does not increase. The counter $N(t, j)$ is subsequently reset to 0, and $\Lambda(t, j) \leftarrow L(t) = 2^j$. Finally, $cache(s_j, t) \leftarrow 2^j$ as well.

Case 2: Otherwise, $i \geq 2 \cdot 2^j$, and the edges (s_j, t) already exist. Instead, insert edges (u_{i-2^j-1}, s_j), for all $s_j \in S_j$. This edge insertion causes $L(s_j)$ to increase to the next multiple of 2^j. After the update, we have $L(s_j) = cache(s_j, t) = \Lambda(t, j) = i - 2^j$, and hence all edges (s_j, t) are followed. The counter $N(t, j)$ again resets to 0, $\Lambda(t, j) \leftarrow L(t) = i$, and finally $cache(s_j, t) \leftarrow i$.

In both cases, whenever the phase number i is a multiple of 2^j, we follow all edges (s_j, t) for all $s_j \in S_j$ and $t \in T$. Consider a fixed j. There are $|S_j| \cdot |T| = 2^{j+2}k$ such edges. Summing over all $k/2^j$ phases during which the phase number is a multiple of 2^j, there are $(2^{j+2}k)(k/2^j) = 4k^2 = \Omega(n^2)$ edge followings from vertices in S_j to vertices in T. Summing over all $\lg(k) - 2 = \Theta(\log n)$ values of j yields a total of $\Omega(n^2 \log n)$ edge followings. □

5 Conclusion

We have shown in this paper how to solve the problem of incremental topological ordering where the total cost of inserting m edges into a graph containing n vertices is $O(n^2 \log n)$. It supports order queries of the form "Does u come before v in the topological ordering?" in $O(1)$ time, and, with minor modifications, it can support successor/predecessor queries in $O(1)$ time. For dense graphs where $m \geq n^{4/3} \log^{2/3} n$, our algorithm is the most efficient to date.

As presented, our algorithm requires $O(n^2)$ space (in terms of machine words, not bits); using a priority queue to manage the outgoing edges should reduce the space to $O(m + n \log n)$ but increase the running time to $O(n^2 \log^2 n)$.

The major open question is whether the new techniques introduced in this paper can yield improvements in the sparse case. For our algorithm, there exists an instance of $n-1$ edge insertions that requires $\Omega(n^2)$ work. For example, always adding edges to the front of a chain results in relabelling every node in the chain on every edge insertion. It would be interesting, however, if some variant of our approach leads to a good algorithm for sparse graphs.

Acknowledgements

We would like to give special thanks to Robert E. Tarjan and the anonymous reviewers for detailed suggestions on how to simplify and improve our algorithm. The current formulation of our algorithm draws heavily on Tarjan's comments, and his suggestions also helped us to improve the running time by a factor of $\log n$.

References

[1] D. Ajwani and T. Friedrich. Average-case analysis of online topological ordering. In T. Tokuyama, editor, *ISAAC*, volume 4835 of *Lecture Notes in Computer Science*, pages 464–475. Springer, 2007.

[2] D. Ajwani, T. Friedrich, and U. Meyer. An $O(n^{2.75})$ algorithm for online topological ordering. In *Proceedings of the 10th Scandinavian Workshop on Algorithm Theory*, volume 4059 of *Lecture Notes in Computer Science*, pages 53–64. Springer, 2006.

[3] B. Alpern, R. Hoover, B. K. Rosen, P. F. Sweeney, and F. K. Zadeck. Incremental evaluation of computational circuits. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 32–42, 1990.

[4] F. Belik. An efficient deadlock avoidance technique. *IEEE Transactions. on Computers*, 39(7), 1990.

[5] M. A. Bender, R. Cole, E. D. Demaine, M. Farach-Colton, and J. Zito. Two simplified algorithms for maintaining order in a list. In *Proceedings of the European Symposium on Algorithms*, pages 152–164, 2002.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. MIT Press, 2001.

[7] P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the ACM Symposium on the Theory of Computing*, pages 365–372, May 1987.

[8] B. Haeupler, T. Kavitha, R. Mathew, S. Sen, and R. E. Tarjan. Faster algorithms for incremental topological ordering.

In *Proceedings of the 35th International Colloquium on Automata, Languages, and Programming*, July 2008.

[9] B. Haeupler, S. Sen, and R. E. Tarjan. Incremental topological ordering and strong component maintenance. *CoRR*, abs/0803.0792, 2008.

[10] I. Katriel. On algorithms for online topological ordering and sorting. Technical Report MPI-I-2004-1-003, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 2004.

[11] I. Katriel and H. L. Bodlaender. Online topological ordering. In *Proceedings of the 16th ACM-SIAM Symposium on Discrete Algorithms*, pages 443–450, Vancouver, British Columbia, Canada, January 2005.

[12] T. Kavitha and R. Mathew. Faster algorithms for online topological ordering. *CoRR*, abs/0711.0251, 2007.

[13] H.-F. Liu and K.-M. Chao. A tight analysis of the katriel–bodlaender algorithm for online topological ordering. *Theoretical Computer Science*, 389(1-2):182–189, 2007.

[14] A. Marchetti-Spaccamela, U. Nanni, and H. Rohnert. On-line graph algorithms for incremental compilation. In *Proceedings of the 19th International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 70–86, June 1993.

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

[16] S. M. Omohundro, C. Lim, and J. Bilmes. The sather language compiler/debugger implementation. Technical report, International Computer Science Institute, Berkeley, March 1992.

[17] D. Pearce, P. Kelly, and C. Hankin. Online cycle detection and difference propagation for pointer analysis. In *Proceedings of the 3rd International Workshop on Source Code Analysis and Manipulation*, pages 3–12, Sept. 2003.

[18] D. J. Pearce and P. H. J. Kelly. A dynamic algorithm for topologically sorting directed acyclic graphs. In *Proceedings of the 3rd International Workshop on Efficient Experimental Algorithms*, volume 3059 of *Lecture Notes in Computer Science*, pages 383–398. Springer, 2004.

[19] G. Ramalingam and T. Reps. On competitive on-line algorithms for the dynamic priority-ordering problem. *Information Processing Letters*, 51(3):155–161, 1994.