A NEW APPROACH TO INCREMENTAL CYCLE DETECTION AND RELATED PROBLEMS

MICHAEL A. BENDER*, JEREMY T. FINEMAN†, SETH GILBERT‡, AND ROBERT E. TARJAN§

Abstract. We consider the problem of detecting a cycle in a directed graph that grows by arc insertions, and the related problems of maintaining a topological order and the strong components of such a graph. For these problems we give two algorithms, one suited to sparse graphs, the other to dense graphs. The former takes $O(\min\{m^{1/2}, n^{2/3}\}m)$ time to insert m arcs into an n-vertex graph; the latter takes $O(n^2 \log n)$ time. Our sparse algorithm is substantially simpler than a previous $O(m^{3/2})$-time algorithm; it is also faster on graphs of sufficient density. The time bound of our dense algorithm beats the previously best time bound of $O(n^{5/2})$ for dense graphs. Our algorithms rely for their efficiency on vertex numberings weakly consistent with topological order: we allow ties. Bounds on the size of the numbers give bounds on running time.

Key words. topological ordering, cycle detection, strongly connected components, incremental data structure

AMS subject classifications. 68P05, 68Q25, 68R10, 68W05, 68W40, 05C20, 05C38, 05C85

1. Introduction. Perhaps the most basic algorithmic problem pertaining to directed graphs is cycle detection. We consider an incremental version of this problem: given an initially empty graph that grows by on-line arc insertions, report the first insertion that creates a cycle. We also consider two related problems, that of maintaining a topological order of an acyclic graph as arcs are inserted, and maintaining the strong components of such a graph.

We use the following terminology. We denote a list by square brackets around its elements; "[ ]" denotes the empty list. We denote list catenation by "&". In a directed graph, we denote an arc from v to w by (v, w). We disallow multiple arcs and loops (arcs of the form (v, v)). We assume that the set of vertices is fixed and known in advance, although our results extend to handle on-line vertex insertions. We denote by n and m the number of vertices and arcs, respectively. We assume that m is known in advance; our results extend to handle the alternative. To simplify expressions for bounds we assume $n > 1$ and $m = \Omega(n)$; both are true if there are no isolated vertices. If (v, w) is an arc, v is a ***predecessor*** of w, and w is a ***successor*** of v. The ***size*** $size(w)$ of a vertex w is the number of vertices v such that there is a path from v to w. Two vertices, two arcs, or a vertex and an arc are ***related*** if they are on a common path, ***mutually related*** if they are on a common cycle (not necessarily simple), and ***unrelated*** if they are not on a common path. Relatedness is a symmetric relation. The ***strong components*** of a directed graph are the subgraphs induced by the maximal subsets of mutually related vertices.

*Stony Brook University. This author's work was supported in part by NSF grants CCF-0621439/0621425, CCF-0540897/05414009, CCF-0634793/0632838, and CNS-0627645

†Georgetown University. This author's work was supported in part by NSF grants CCF-0621511, CNS-0615215, CCF-0541209, and NSF/CRA sponsored CIFellows program.

‡National University of Singapore.

§Princeton University and HP Labs. This author's work was supported in part by NSF grants CCF-0830676 and CCF-0832797, the U.S.-Israel Binational Science Foundation grant 2006204 and by the Distinguished Visitor program in the Stanford University Computer Science Department.

The $O(n^2 \log n)$ algorithm for dense graphs previously appeared in [6], but the other algorithm and strong-component extensions are new to this paper.

A ***DAG*** is a directed acyclic graph. A ***topological order*** of a DAG is a total order $<$ on the vertices such that if (v, w) is an arc, $v < w$. A ***topological numbering*** of a DAG is a numbering of the vertices from 1 through n such that increasing numeric order is a topological order. A ***weak topological numbering*** of a DAG is a numbering of the vertices such that if (v, w) is an arc, v is numbered less than w. A ***pseudo topological numbering*** of a DAG is a numbering of the vertices such that if (v, w) is an arc, v is numbered no greater than w. In either a weak or pseudo topological numbering, the vertex numbers can be arbitrary, and several vertices can have the same number. A topological numbering is a weak topological numbering; a weak topological numbering is a pseudo topological numbering.

There has been much recent work on incremental cycle detection, topological ordering, and strong component maintenance [1, 2, 3, 4, 8, 9, 12, 13, 16, 17, 18, 21, 22]. For a thorough discussion of this work see [9]; here we discuss the heretofore best results and others related to our work. A classic result of graph theory is that a directed graph is acyclic if and only if it has a topological order [25]; a more recent generalization is that the strong components of a directed graph can be ordered topologically (so that every arc lies within a component or leads from a smaller component to a larger one) [10]. For static graphs, there are two $O(m)$-time algorithms to find a cycle or a topological order: repeated deletion of vertices with no predecessors [11, 14, 15] and depth-first search [26]: the reverse postorder [27] defined by such a search is a topological order if the graph is acyclic. Depth-first search extends to find the strong components and a topological order of them in $O(m)$ time [26]

For the problems of incremental cycle detection, topological ordering, and strong component maintenance, there are two known fastest algorithms, one suited to sparse graphs, the other suited to dense graphs. Both are due to Haeupler et al. [8, 9]. Henceforth we denote the coauthors of these papers by HKMST. The HKMST sparse algorithm takes $O(m^{3/2})$ time for m arc additions; the HKMST dense algorithm takes $O(n^{5/2})$ time. Both of these algorithms use two-way search; each is a faster version of an older algorithm. These algorithms, and the older ones on which they are based, bound the total running time by counting the number of arc pairs or vertex pairs that become related as a result of arc insertions. The HKMST sparse algorithm uses a complicated dynamic list data structure [5, 7] to represent a topological order, and it uses either linear-time selection or random sampling to guide the searches. There are examples on which the algorithm takes $\Omega(nm^{1/2})$ time, so its time bound is tight for sparse graphs. The time bound of the HKMST dense algorithm is not known to be tight, but there are examples on which it takes $\Omega(n^2 2^{\sqrt{2 \lg n}})$ time [9].

Our approach to incremental cycle detection and the related problems is different. We maintain a pseudo or weak topological numbering and use it to facilitate cycle detection. Our algorithms pay for cycle-detecting searches by increasing the numbers of appropriate vertices; a bound on the numbers gives a bound on the running time. One insight is that the size function is a weak topological numbering. Unfortunately, maintaining this function as arcs are inserted seems to be expensive. But we are able to maintain in $O(n^2 \log n)$ time a weak topological numbering that is a lower bound on size. This gives an incremental cycle detection algorithm with the same running time, substantially improving the time bound of the HKMST dense algorithm. Our algorithm uses one-way rather than two-way search. For sparse graphs, we use a pseudo topological numbering. This idea yields a very simple algorithm with a running time of $O(\min\{m^{1/2}, n^{2/3}\}m)$. Our algorithm is substantially simpler than the HKMST sparse algorithm and asymptotically faster on sufficiently dense graphs.

The $O(n^2 \log n)$ algorithm appeared previously in [6], but the other algorithm is new to this paper.

The remainder of our paper consists of four sections. Section 2 describes our cycle-detection algorithm for sparse graphs. Section 3 describes our cycle-detection algorithm for dense graphs. Section 4 describes several simple extensions of the algorithms. Section 5 extends the algorithms to maintain the strong components of the graph as arcs are inserted instead of stopping as soon as a cycle exists. The extensions in Sections 4 and 5 preserve the asymptotic time bounds of the algorithms. Section 6 contains concluding remarks.

2. A Two-Way-Search Algorithm for Sparse Graphs. Our algorithm for sparse graphs uses two-way search to look for cycles. Unlike the entirely symmetric forward and backward searches in the HKMST sparse algorithm, the two searches in our algorithm have different functions. Also unlike the HKMST sparse algorithm, our algorithm avoids the use of a dynamic list data structure, and it does not use selection or random sampling: all of its data structures are very simple, as is the algorithm itself.

We maintain a pseudo topological numbering. This numbering partitions the vertices into levels. Each backward search proceeds entirely within a level. If the search takes too long, we stop it and increase the level of a vertex. This bounds the backward search time. Each forward search traverses only arcs that lead to a lower level, and it increases the level of each vertex visited. An overall bound on such increases gives a bound on the time of all the forward searches.

Here are the details. Each vertex v has a positive integer ***level*** $k(v)$. The levels are a pseudo topological order. For each vertex v, we maintain the set $out(v)$ of outgoing arcs (v, w) (to facilitate forward search) and the set $in(v)$ of incoming arcs (u, v) such that $k(u) = k(v)$ (to facilitate backward search). Initially $k(v) = 1$ for all vertices, and all incident arc sets are empty. Let $\Delta = \min\{m^{1/2}, n^{2/3}\}$. The algorithm for adding a new arc (v, w) consists of the following four steps:

Step 1 (test order): If $k(v) < k(w)$, go to Step 4 (the levels remain a pseudo topological numbering).

Step 2 (search backward): Using the incoming arc sets, search backward from v, visiting only vertices on the same level, until one of the following occurs: w is visited, at least Δ arcs are traversed, or no backward arcs remain to be traversed. Let B be the set of visited vertices. If w is visited, stop and report a cycle. If the search completes without traversing at least Δ arcs and $k(w) = k(v)$, go to Step 4 (the levels remain a pseudo topological ordering). If the search completes without traversing at least Δ arcs and $k(w) < k(v)$, set $k(w) = k(v)$. If the search traverses at least Δ arcs, set $k(w) = k(v) + 1$ and $B = \{v\}$. In either of the last two cases (those in which $k(w)$ increases), set $in(w) = \{\}$ and continue to Step 3.

Step 3 (search forward): Using the outgoing arc sets, search forward from w, following outgoing edges only from vertices whose level increases, until a vertex in B is visited or no forward arcs remain to be traversed. The forward search updates the incoming arc sets as vertex levels increase. Specifically, when traversing a forward arc (x, y), if $y \in B$, stop and report a cycle. If $k(x) = k(y)$, add (x, y) to $in(y)$. If $k(x) > k(y)$, set $k(y) = k(x)$, set $in(y) = \{(x, y)\}$, and add all arcs in $out(y)$ to those to be traversed.

Step 4 (insert arc): Add (v, w) to $out(v)$. If $k(v) = k(w)$, add (v, w) to $in(w)$.

THEOREM 2.1. *While the graph remains acyclic, the levels are a pseudo topological numbering and the incident arc sets are correct. The algorithm stops and reports a cycle if and only if the last arc insertion creates a cycle.*

Proof. We prove the theorem by induction on the number of arc insertions. The theorem holds before any arcs are inserted. Suppose the theorem holds just before the insertion of arc (v, w). If there is a path from w to v, then all vertices on it, including w, have level at most $k(v)$, since levels are a pseudo topological numbering. Thus if $k(v) < k(w)$, there is no path from w to v, the addition of (v, w) does not create a cycle, the levels remain a pseudo topological numbering after the insertion of (v, w), the algorithm correctly updates the arc sets in Step 4, and the theorem holds after (v, w) is added.

Suppose on the other hand that $k(v) \geq k(w)$. If the algorithm visits w during the backward search, or visits some vertex in B during the forward search, then there is a path from w to v. This path forms a cycle with arc (v, w). Thus, if the algorithm stops and reports a cycle, there is one.

Suppose the insertion of (v, w) creates a cycle. Then there is a path P from w to v before the insertion of (w, v). If $k(v) = k(w)$, then all vertices on the path from w to v have level $k(v)$. Either the search backward from v visits w and reports a cycle, or the search stops before visiting w, which it can only do after traversing at least Δ arcs. In this case, it increases the level of w to $k(v) + 1$ and begins a forward search. We claim that the forward search stops and reports a cycle. Suppose not. Then there must be an untraversed arc on P. Let (x, y) be the first such arc on P. Then $x \neq w$, since all arcs out of w are traversed. When x is first visited, its level is less than $k(w)$, so the visit causes (x, y) to be traversed eventually. This contradiction establishes the claim.

Suppose on the other hand that $k(w) < k(v)$. If the backward search traverses at least Δ arcs, then it increases the level of w to $k(v) + 1$, and the forward search stops and reports a cycle by the argument in the previous paragraph. Suppose the backward search finishes before traversing at least Δ arcs. Let B be the set of vertices visited by the backward search. After the backward search, the level of w increases to $k(v)$. The first part of P is a path from w through zero or more vertices of level less than $k(v)$ to a vertex in B. An argument like that in the previous paragraph shows that the forward search will traverse every arc on this path, visit a vertex in B, and report a cycle, unless it stops and reports another cycle before this happens. Thus, if the insertion of (v, w) creates a cycle, the algorithm stops and reports one.

Suppose the insertion of (v, w) does not create a cycle. If the backward search finishes before traversing at least Δ arcs and $k(v) = k(w)$, then no vertex increases in level, the levels remain a pseudo topological numbering, and the algorithm correctly updates the incident arc sets in Step 4. If the backward search finishes before traversing at least Δ arcs but $k(v) > k(w)$, or if the backward search traverses at least Δ arcs, then w and possibly other vertices increase in level, to $k(v)$ in the former case, to $k(v) + 1$ in the latter. Let F be the set of vertices whose level increases. If (x, y) is an arc with $x \in F$, then the forward search traverses (x, y), after which $k(x) \leq k(y)$. It follows that after the forward search, the levels are a pseudo topological numbering.

Step 4 adds (v, w) to $out(v)$, and to $in(w)$ if $k(v) = k(w)$, thus correctly updating the incident arc sets to reflect the insertion of (v, w). All that remains is to show that

the algorithm correctly updates the incoming arc sets to reflect increases in vertex levels. Let (x, y) be an arc other than (v, w) such that x or y increases in level. If y increases in level but x does not, then $k(x) < k(y)$ after the insertion of (v, w), the increase in $k(y)$ deletes (x, y) from $in(y)$ if it were there, and (x, y) is not traversed by the forward search, so it is not later added to $in(y)$. If x increases in level, (x, y) is traversed by the forward search. If y does not increase in level, then (x, y) is not in $in(y)$ before the insertion of (v, w) and is added to $in(y)$ by the traversal of (x, y) if and only if the new level of x is that of y. If y increases in level as a result of the traversal of (x, y), then the traversal correctly adds (x, y) to $in(y)$. If y increases in level as a result of some other event, then the increase deletes (x, y) from $in(y)$ if it were there; the traversal of (x, y) correctly adds (x, y) to $in(y)$. Thus, the algorithm correctly maintains the incoming arc sets. □

LEMMA 2.2. *No vertex level exceeds* $\min\left\{m^{1/2}, n^{2/3}\right\} + 2$.

Proof. Fix a topological order just before the last arc insertion. Let $k > 1$ be a level assigned before the last arc insertion, and let w be the lowest vertex in the fixed topological order assigned level k. For w to be assigned level k, the insertion of an arc (v, w) must cause a backward search from v that traverses at least Δ arcs both ends of which are on level $k-1$. All the ends of these arcs must still be on level $k-1$ just before the last insertion. Thus these sets of arcs are distinct for each k, as are their sets of ends. Since there are only m arcs, there are most m/Δ distinct values of k. Also, for each k there must be at least $\sqrt{\Delta}$ distinct arc ends, since there are no loops or multiple arcs. Since there are only n vertices, there are at most $n/\sqrt{\Delta}$ distinct values of k. It follows that no vertex level exceeds $\min\left\{m/\Delta, n/\sqrt{\Delta}\right\} + 2$, which gives the lemma. □

The space required by the algorithm is $\Theta(m)$. The next two theorems show that the worst-case time for m arc insertions is $\Theta(\Delta m)$.

THEOREM 2.3. *The algorithm takes* $O(\min\left\{m^{1/2}, n^{2/3}\right\} m)$ *time for* m *arc insertions.*

Proof. Each backward search takes $O(\Delta) = O(\min\left\{m^{1/2}, n^{2/3}\right\})$ time. The time spent adding and removing arcs from incidence sets is $O(1)$ per arc added or removed. An arc can be added or removed only when it is inserted into the graph or when the level of one of its ends increases. By Lemma 2.2, this can happen at most $O(\min\left\{m^{1/2}, n^{2/3}\right\})$ times per arc. The time for a forward search is $O(1)$ plus $O(1)$ per arc (x, y) such that x increases in level as the result of the arc insertion that triggers the search. By Lemma 2.2, this happens $O(\min\left\{m^{1/2}, n^{2/3}\right\})$ times per arc. □

THEOREM 2.4. *For any* n *and* m *with* $m \leq n(n-1)/2$*, there exists a sequence of* m *arc insertions causing the algorithm to run in* $\Omega(\min\left\{m^{1/2}, n^{2/3}\right\} m)$ *total time.*

Proof. Assume without loss of generality that $m \geq 2n$ and n is sufficiently large. Let the vertices be 1 through n, numbered in the initial topological order. We first add arcs (i, j) with $i < j$ to construct a number of cliques of consecutive vertices. When adding these arcs, we add them in decreasing order on i, so that each backward search visits no arcs and causes no vertex to increase in level. An ***r-clique*** of vertices k through $k+r-1$ is formed by adding arc (i, j) for i, j such that $k \leq i < j \leq k+r-1$. An r-clique consists of r vertices and $r(r-1)/2$ arcs.

Let $r_1 = \lfloor \sqrt{m}/2 \rfloor$. Construct an r_1-clique of the first r_1 vertices. This is the ***main clique***. The main clique contains at most $n/2$ vertices and at most $m/4$ arcs. Let $r_2 = \left\lceil \sqrt{\Delta} + 1 \right\rceil$. Starting with vertex $r_1 + 1$, construct r_2-cliques on disjoint sets of consecutive vertices, until running out of vertices or until $\lfloor m/2 \rfloor$ arcs have been added, including those added to make the main clique. Each of the r_2-cliques is an ***anchor clique***. The number of arcs in each anchor clique is $O(\Delta)$ and at least Δ. Number the anchor cliques from 1 though k. Then $k = \Theta(\Delta)$. So far all vertices have level 1.

Next, for j from 1 through $k-1$, add arcs from the last vertex of anchor clique j to each vertex of anchor clique $j+1$. Add these arcs in decreasing topological order with respect to the end of the arc that is in anchor clique $j+1$. There are at most $n/2 \leq m/4$ such arc additions. Each addition of an arc from the last vertex of anchor clique 1 to a vertex w in anchor clique 2 triggers a backward search that traverses at least Δ arcs and causes the level of w to increase from 1 to 2. Each forward search visits only a single vertex. Once all arcs from anchor clique 1 are added, all vertices in anchor clique 2 have level 2. Addition of the arcs from the last vertex of anchor clique 2 to the vertices in anchor clique 3 moves all vertices in anchor clique 3 to level 3. After all the arcs between anchor cliques are added, every vertex in anchor clique j is on level j. The number of arcs added to obtain these level increases is at most $n/2 \leq m/4$.

Finally, for each anchor clique from 2 through k add an arc from its first vertex in topological order to the first vertex in the main clique. There are at most $n/2 \leq m/4$ such arc additions. Each addition triggers a backward search that visits only one vertex, followed by a forward search that traverses all the arcs in the main clique and increases the level of all vertices in the main clique by one. These forward searches do $\Theta(\Delta m)$ arc traversals altogether. At most m arcs are added during the entire construction. □

We can extend the algorithm to maintain a weak topological numbering by breaking ties within levels in a way consistent with a topological order. To do this we assign each vertex v an integer ***index*** $i(v)$ as well as a level, and combine the level and index of a vertex into a single number. To update the indices efficiently, we make the backward and forward searches depth-first.

Here are the details. Let $a = b = nm+1$. Initialize $k(v) = 1$ and $i(v) = a$ for each vertex v. The algorithm maintains the invariant that the numbering $bk(v) + i(v)$ is a weak topological numbering. Variable a counts down and is used to update indices. The algorithm for adding a new arc (v, w) consists of the following five steps:

Step 1 (test order): If $bk(v) + i(v) < bk(w) + i(w)$, go to Step 5 (the numbering remains a weak topological numbering).

Step 2 (search backward): Using the incoming arc sets, do a depth-first search backward from v, visiting only vertices on the same level, until visiting w, traversing at least Δ arcs, or running out of arcs to traverse. Let B be a list of the visited vertices in postorder with respect to the search (thus a vertex appears later in B than all of its predecessors). If w is visited, stop and report a cycle. If the search stops without traversing at least Δ arcs and $k(w) = k(v)$, set $L = B$ and go to Step 4 (the levels remain a pseudo topological ordering). If the search stops without traversing at least Δ arcs and $k(w) < k(v)$, set $k(w) = k(v)$. If the search traverses at least Δ

arcs, set $k(w) = k(v) + 1$ and $B = [v]$. In either of the last two cases (where $k(w)$ increases), set $in(w) = \{\}$ and continue to Step 3.

Step 3 (search forward): Using the outgoing arc sets, do a depth-first search forward from w, following outgoing edges only from vertices whose level increases, stopping early if a vertex in B is visited. Let F be a list of the vertices whose level increases in reverse postorder with respect to the search (thus a vertex appears earlier in F than all of its successors). When traversing a forward arc (x, y), if $y = v$ or $y \in B$, stop and report a cycle. If $k(x) = k(y)$, add (x, y) to $in(y)$. If $k(x) > k(y)$, set $k(y) = k(x)$, set $in(y) = \{(x, y)\}$, and traverse all arcs in $out(y)$. If the forward search finishes without detecting a cycle, set $L = F$ if $k(v) < k(w)$ or $L = B\&F$ if $k(v) = k(w)$, and continue to Step 4.

Step 4 (update indices): While L is non-empty, set $a = a-1$, delete the last vertex x on L, and set $i(x) = a$.

Step 5 (insert arc): Add (v, w) to $out(v)$. If $k(v) = k(w)$, add (v, w) to $in(w)$.

THEOREM 2.5. *While the graph remains acyclic, the vertex numbering is a weak topological numbering and the incident arc sets are correct. The extended algorithm stops and reports a cycle if and only if the last arc insertion creates a cycle.*

Proof. The set L in Step 4 contains at most n vertices, since each vertex can be on B or F but not both. Thus a remains positive over all m arc additions, and $k(v) < k(w)$ implies $bk(v) + i(v) < bk(w) + i(w)$. The proof is the same as the proof of Theorem 2.1, except that we must show that Step 4 guarantees that the new numbering is a weak topological numbering after the arc addition. Suppose the insertion of (v, w) triggers renumbering. After the renumbering, $k(v) \leq k(w)$. If $k(v) = k(w)$, then $v \in B$. Whether or not a forward search occurs, if $k(v) = k(w)$ then v gets a new index smaller than that of w. Thus, $bk(v) + i(v) < bk(w) + i(w)$. Let (x, y) be an arc other than (v, w). After the renumbering, $k(x) \leq k(y)$ by the proof of Theorem 2.1. Suppose $k(x) = k(y)$. If $y \in L$, x must be in L, so $i(y) < i(x)$ after the renumbering: B is in postorder with respect to the backward search, if F is defined it is in reverse postorder with respect to the forward search, and no arc leads from F to B. If x but not y is in L, then $i(x) < i(y)$ after the renumbering since every new index is smaller than every old index. If neither x nor y is in L, then neither is renumbered. It follows that the new numbering is a weak topological order. □

3. A One-Way-Search Algorithm for Dense Graphs. The two-way-search algorithm becomes less and less efficient as the graph density increases. For sufficiently dense graphs, the one-way search algorithm we present in this section is better: it takes $O(n^2 \log n)$ time for any number (up to $n(n-1)$) of arc insertions. The algorithm maintains for each vertex v a level $k(v)$ that is a weak topological numbering satisfying $k(v) \leq size(v)$. The algorithm pays for its searches by increasing vertex levels, using the following lemma to maintain $k(v) \leq size(v)$ for all v.

LEMMA 3.1. *In an acyclic graph, if a vertex v has j predecessors, each of size at least s, then $size(v) \geq s + j$.*

Proof. Order the vertices of the graph in topological order and let u be the smallest predecessor of v. Then $size(v) \geq size(u) + j \geq s + j$. Here "$+j$" counts v and the $j - 1$ predecessors of v other than u. □

The algorithm uses Lemma 3.1 on a hierarchy of scales. For each vertex v, in addition to a level $k(v)$, it maintains a bound $b_i(v)$ and a count $c_i(v)$ for each integer $i, 0 \leq i \leq \lfloor \lg n \rfloor$, where lg is the base-2 logarithm. Initially $k(v) = 1$ for all v, and $b_i(v) = c_i(v) = 0$ for all v and i. To represent the graph, for each vertex v the algorithm stores the set of outgoing arcs (v, w) in a heap (priority queue) $out(v)$, each arc having a ***priority*** that is at most $k(w)$. (This priority is either $k(w)$ or a previous value of $k(w)$.) Initially all such heaps are empty.

The arc insertion algorithm maintains a set of arcs A to be traversed, initially empty. To insert an arc (v, w), add (v, w) to A and repeat the following step until a cycle is detected or A is empty:

Traversal Step:

delete some arc (x, y) from A
if $y = v$
  then stop the algorithm and report a cycle
if $k(x) \geq k(y)$
  then $k(y) \leftarrow k(x) + 1$
  else // $k(x) < k(y)$
    $i \leftarrow \lfloor \lg(k(y) - k(x)) \rfloor$
    $c_i(y) \leftarrow c_i(y) + 1$
    if $c_i(y) = 3 \cdot 2^{i+1}$
      then $c_i(y) \leftarrow 0$
        $k(y) \leftarrow \max\{k(y), b_i(y) + 3 \cdot 2^i\}$
        $b_i(y) \leftarrow k(y) - 2^{i+1}$.
delete from $out(y)$ all arcs with priority at most $k(y)$ and add these arcs to A.
add (x, y) to $out(x)$ with priority $k(y)$.

In a traversal step, an arc (y, z) that is deleted from $out(y)$ may have $k(z) > k(y)$, because $k(z)$ may have increased since (y, z) was last inserted into $out(y)$. Subsequent traversal of such an arc may not increase k(z). It is to pay for such traversals that we need the mechanism of bounds and counts.

We implement each heap $out(v)$ as an array of buckets indexed from 1 through n, with bucket i containing the arcs with priority i. We also maintain the smallest index of a nonempty bucket in the heap. This index never decreases, so the total time to increment it over all deletions from the heap is $O(n)$. The time to insert an arc into a heap is $O(1)$. The time to delete a set of arcs from a bucket is $O(1)$ per arc deleted. The time for heap operations is thus $O(1)$ per arc traversal plus $O(n)$ per heap. Since there are n heaps, this time totals $O(1)$ per arc traversal plus $O(n^2)$.

The space needed by the algorithm is $O(nlogn + m)$ for the labels, bounds, and counts, and $O(n^2)$ for the n heaps. Storing the heaps in hash tables reduces their total space to $O(m)$ but makes the algorithm randomized. By using a two-level data structure [29] to store each heap, the space for the heaps can be reduced to $O(n^{1.5}+m)$ without using randomization. This bound is $O(m)$ if $m/n = \Omega(n^{1/2})$; if not, the sparse algorithm of Section 2 is faster.

To analyze the algorithm, we begin by bounding the total number of arc traversals, thereby showing that the algorithm terminates. Then we prove its correctness. Finally, we bound the running time.

LEMMA 3.2. *While the graph remains acyclic, the insertion algorithm maintains*

$k(v) \leq size(v)$ *for every vertex* v.

Proof. The proof is by induction on the number of arc insertions. The inequality holds initially. Suppose it holds just before the insertion of an arc (v, w) that does not create a cycle. Consider a traversal step during the insertion that deletes (x, y) from A and increases $k(y)$. If $k(y)$ increases to $k(x) + 1$, $size(y) \geq 1 + size(x) \geq 1 + k(x)$, maintaining the inequality for y. The more interesting case is when $c_i(y) = 3 \cdot 2^{i+1}$ and k(y) increases to $b_i(y) + 3 \cdot 2^i$. Each of the increases to $c_i(y)$ since it was last zero corresponds to the traversal of an arc (z, y). When $c_i(y)$ was last zero, $b_i(y) = \max\left\{0, k(y) - 2^{i+1}\right\}$. Since $k(y)$ cannot decrease, $b_i(y) \leq k(z) \leq size(z)$ when this traversal of (z, y) occurs, since at this time $k(y) - k(z) < \min\left\{k(y), 2^{i+1}\right\}$. We consider two cases. If there were at least $3 \cdot 2^i$ traversals of distinct arcs (z, y) since $c_i(y)$ was last zero, then $size(y) \geq b_i(y) + 3 \cdot 2^i$ by Lemma 3.1, and the increase in $k(y)$ maintains the inequality for y. If not, by the pigeonhole principle there were at least three traversals of a single arc (z, y) since $c_i(y)$ was last zero. When each traversal happens, $k(y) - k(z) \geq 2^i$, but each of the second and third traversals cannot happen until $k(z)$ increases to at least the value of $k(y)$ at the previous traversal. This implies that when the third traversal happens, $k(y) \geq b_i(y) + 3 \cdot 2^i$, so $k(y)$ will not in fact increase as a result of this traversal. □

LEMMA 3.3. *If a new arc* (v, w) *creates a cycle, the insertion algorithm maintains* $k(v) \leq size(v) + n$, *where sizes are before the addition of* (v, w).

Proof. Before the addition of (v, w), $k(v) \leq size(v)$ for every vertex v, by Lemma 3.2. Traversal of the arc (v, w) can increase $k(v)$ by at most n, so the desired inequality holds after this traversal. Every subsequent traversal is of an arc other than (v, w): to traverse (v, w), an arc into v must be traversed, which results in reporting of a cycle. Thus the subsequent traversals are of arcs in the acyclic graph before the addition of (v, w). The proof of Lemma 3.2 extends to prove that these traversals maintain the desired inequality: Lemma 3.1 holds if the size function is replaced by the size plus any constant, in particular by the size plus n. □

LEMMA 3.4. *The total number of arc traversals for* m *arc additions is* $O(n^2 \log n)$.

Proof. By Lemmas 3.2 and 3.3, every label $k(v)$, and hence every bound $b_i(v)$, remains below $2n$. Every arc traversal increases a vertex level or increases a count. The number of level increases is $O(n^2)$. Consider a count $c_i(v)$. Each time $c_i(v)$ is reset to zero from $3 \cdot 2^{i+1}$, $b_i(v)$ increases by at least 2^i. Since $b_i(v) \leq 2n$, the total amount by which $c_i(v)$ can decrease as a result of being reset is at most $12n$. Since $c_i(v)$ starts at zero and cannot exceed $4n$, the total number of times $c_i(v)$ increases is at most $16n$. Summing over all counts for all vertices gives a bound of $O(n^2 \log n)$ on the number of count increases and hence on the number of arc traversals. □

THEOREM 3.5. *If the insertion of an arc* (v, w) *creates a cycle, the insertion algorithm stops and reports a cycle. If not, the insertion algorithm maintains the invariant that* k *is a weak topological numbering.*

Proof. By Lemma 3.4 the algorithm terminates. A straightforward induction shows that every arc (x, y) traversed by the insertion algorithm is such that x is reachable from v, so if the algorithm stops and reports a cycle, there is one. Suppose the insertion of (v, w) creates a cycle. Before the insertion of (v, w), k is a weak topological numbering, so the path from w to v existing before the addition of (v, w)

has vertices in strictly increasing order. Thus v has the largest level on the path. A straightforward induction shows that the algorithm will eventually traverse every arc on the path and report a cycle, unless it reports another cycle first.

Suppose addition of an arc (v, w) does not create a cycle. Before the addition, k is a weak topological numbering. The algorithm maintains the invariant that every arc (x, y) such that $k(x) \geq k(y)$ is either on A or is the arc being processed. Thus once A is empty, k is a weak topological numbering. □

THEOREM 3.6. *The algorithm runs in $O(n^2 \log n)$ total time.*

Proof. The running time is $O(1)$ per arc traversal plus $O(n^2)$. This is $O(n^2 \log n)$ by Lemma 3.4. □

The following result shows that the bound in Theorem 3.6 is tight.

THEOREM 3.7. *For any sufficiently large n, there exists a sequence of $\Theta(n^2)$ arc insertions that causes the algorithm to do $\Omega(n^2 \log n)$ arc traversals.*

Proof. Without loss of generality, suppose $n = (7/2)r - 3$, where $r \geq 2^3$ is a power of 2. The graph we construct consists of three categories of vertices: (1) vertices $u_1, u_2, \ldots, u_r$, (2) sets of vertices $S_0, S_1, \ldots, S_{\lg(r)-2}$ with $|S_j| = 3 \cdot 2^{j+1}$ (so $\sum_j |S_j| = 3(r/2 - 1)$), and (3) a set of vertices T with $|T| = r$. Initially there are no arcs in the graph, and all levels are 1.

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

The arc insertions proceed in phases ranging from 2 to r. In phase i, first insert arc (u_{i-1}, t) for all $t \in T$, thereby increasing $k(t)$ to i. Next, consider each j for which there exists a constant $c \geq 3$ such that $i = c2^j$, i.e., i is a sufficiently large multiple of 2^j. There are two cases here, described in more detail shortly. If $c = 3$, insert arcs from S_j to T, not causing a level increase to t. If $c > 3$, the algorithm traverses the arcs from S_j to T again, but without causing any level increases to $t \in T$. Moreover, the only time any $c_j(t)$ or $b_j(t)$ changes, for $j > 0$, is when the algorithm traverses an arc from S_j to $t \in T$.

Case 1 (add arcs from S_j to T): If $i = 3 \cdot 2^j$ for some j, add arcs $(u_{2^{j+1}-1}, s_j)$ for all $s_j \in S_j$, causing $k(s_j)$ to increase to 2^{j+1}. Also add arcs (s_j, t) for all $s_j \in S_j$ and $t \in T$. Observe that before these arc additions $\lfloor \lg(k(t) - k(s_j)) \rfloor = \lfloor \lg(2^{j+1} - 2^j) \rfloor = j$. Moreover, $c_j(t) = 0$ and $b_j(t) = 0$. For each t, when the last arc insertion occurs, $c_j(t)$ increases to $3 \cdot 2^{j+1}$. We have, however, that $k(t) = 3 \cdot 2^{j+1} > b_j + 3 \cdot 2^j$, and hence $k(t)$ does not increase. The counter $c_j(t)$ is subsequently reset to 0 and $b_j(t) = k(t) - 2^{j+1} = k(s_j) - 2^j$. Finally, the priority of each of these arcs (s_j, t) is updated to $3 \cdot 2^j$ in $out(s_j)$.

Case 2 (follow arcs from S_j to T): Otherwise, $i = c2^j$, for $c > 3$. Since $i > 3 \cdot 2^j$, the arcs (s_j, t) already exist. Before this step, we have $k(s_j) = k(t) - 2^{j+1}$, for each $s_j \in S_j$. Moreover, we have $b_j(t) = k(s_j) - 2^j = k(t) - 3 \cdot 2^j$. Insert arcs (u_{i-2^j-1}, s_j), for all $s_j \in S_j$. Such an arc insertion causes $k(s_j)$ to increase to the next multiple of 2^j. After the update, we have $k(s_j)$ equal to the priority of each arc (s_j, t) in $out(s_j)$, and hence the algorithm traverses each of the outgoing arcs. Moreover, $\lg(k(t) - k(s_j)) = \lg(i - 2^j) = j$, and hence the counter c_j is affected.

For each t, the counter $c_j(t)$ again reaches $3 \cdot 2^{j+1}$. Since $b_j(t) = k(t) - 3 \cdot 2^j$, the level of t again does not increase. The counter $c_j(t)$ is subsequently reset to 0, each $b_j(t) = k(t) - 2^{j+1} = k(s_j) - 2^j$, and the priority of each of the arcs (s_j, t) is set to $k(t)$ in $out(s_j)$.

In both cases, whenever the phase number i is a large enough multiple of 2^j, the algorithm traverses all arcs (s_j, t) such that $s_j \in S_j$ and $t \in T$. Consider a fixed j. There are $|S_j| \cdot |T| = 3 \cdot 2^{j+1} r$ such arcs. Summing over all $r/2^j - 2$ phases during which the phase number is a large enough multiple of 2^j, there are $(3 \cdot 2^{j+1} r)(r/2^j - 2) = \Omega(r^2) = \Omega(n^2)$ arc traversals from vertices in S_j to vertices in T. Summing over all $\lg(r) - 2 = \Theta(\log n)$ values of j yields a total of $\Omega(n^2 \log n)$ arc traversals. □

The proof of Theorem 3.7 extends to give a slightly more general result: for any $1 \leq k \leq \lg n$, there is a sequence of $\Theta(2^k n)$ arc insertions causing the algorithm to do $\Theta(n^2 k)$ arc traversals. To prove this, omit from the proof of Theorem 3.7 the sets S_j with $j > k$. The generalization implies that $\Theta(n)$ arcs are enough to make the algorithm take $\Omega(n^2)$ time, and $\Theta(n^{1+\epsilon})$ arcs, for any constant $\epsilon > 0$, are enough to make the algorithm take $\Omega(n^2 \log n)$ time.

4. Simple Extensions. In this section we extend our sparse and dense algorithms to provide some additional capabilities possessed by previous algorithms. All the extensions are simple and preserve the asymptotic time bounds of the unextended algorithms.

Our first extension eliminates ties of vertex numbers. To eliminate ties in the extended sparse algorithm of Section 2 (which maintains a weak topological numbering), we set $b = nm+n+1$ (instead of $b = nm+1$), and we initially assign the vertices distinct indices from $nm + 1$ to $nm + n + 1$ (instead of initializing all indices to 1). Then all indices remain distinct between 1 and $nm + n + 1$, so all vertex numbers remain distinct. We can eliminate ties in the dense algorithm Section 3 in the same way: set $b = nm + n + 1$, set $a = nm + 1$, give each vertex v a distinct index $i(v)$ between $nm+1$ and $nm+n+1$ inclusive, and define the number of v to be $bk(v)+i(v)$; when a vertex v increases in level, decrement a and set $i(v) = a$.

This way of breaking ties is more complicated than necessary for the dense algorithm (we could just set $b = n$ and give the vertices fixed distinct indices between 1 and n inclusive), but it facilitates our second extension, which maintains a doubly linked list of the vertices in increasing lexicographic order by level and index, and hence in a topological order. The method works for either the sparse or dense algorithm. We maintain a pointer to the first vertex on the list. We also maintain, for each level j, a pointer to the first vertex of level j or higher, if any. When a vertex decreases in index, it becomes the new first vertex in its current level, which may be the same or higher than its old level. Moving a vertex and making all needed pointer changes takes time proportional to one plus the amount by which the vertex level increases, and hence does not affect the asymptotic running time. When moving a group of vertices whose indices change as a result of an arc insertion, we move them in decreasing order by new index, which is the same as the order in which the new indices are assigned.

Our third extension explicitly returns a cycle when one is discovered, rather than just reporting that one exists. We augment each search to grow a spanning tree represented by parent pointers as each search proceeds. In the sparse algorithm, the backward search generates an in-tree rooted at v containing all visited vertices; the forward search generates an out-tree rooted at w containing all vertices whose

level increases. If the backward search causes $k(w)$ to increase to $k(v) + 1$ and B to become empty, the forward search may visit vertices previously visited by the backward search. Each such vertex acquires a new parent when the forward search visits it for the first time. When the algorithm stops and reports a cycle, a cycle can be obtained explicitly by following parent pointers. Specifically, if the backward search traverses an arc (w, y), following parent pointers from y gives a path from y to v, which forms a cycle with (v, w) and (w, y). If the forward search traverses an arc (x, y) with $y = v$ or y in B, traversing parent pointers from x and from y gives a path from w to x and a path from y to v, which form a cycle with (x, y) and (v, w). In the dense algorithm, there is only one tree, an out-tree rooted at v, containing v and all vertices whose level increases. Vertex v has one child, w. If the search traverses an arc (x, v), following parent pointers from x gives a path from v through (v, w) to x, which forms a cycle with (x, v).

Our fourth extension is to handle vertex insertions and to allow n and m to be unknown in advance. We maintain n and m as vertex and arc insertions occur. In the unextended sparse algorithm we give each new vertex an initial level of 1, and we update Δ each time n or m increases. In the unextended dense algorithm, we also give each new vertex an initial level of 1, and each time $\lfloor \lg n \rfloor$ increases, we add a corresponding new set of bounds and constants. If we want to add indices to either algorithm, we maintain distinct indices as well as a doubly linked list of the vertices in increasing lexicographic order by level and index. Until the first arc is inserted, we give new vertices initial indices in increasing order starting from 1. When the first arc is inserted, we initialize $b = 8n^3 + 2n + 1$, $a = 8n^3 + n + 1$, and give the vertices distinct indices between $8n^3 + n + 1$ and $8n^3 + 2n + 1$ inclusive. Subsequently, when a new vertex v is inserted we decrement a and set $i(v) = a$. Each time n doubles, we re-initialize a and b and assign new vertex indices from $8n^3 + n + 1$ to $8n^3 + 2n + 1$ inclusive, in increasing order with respect to the current list order. The extra overhead for re-initializing indices is $O(n)$.

5. Maintenance of Strong Components. A less straightforward extension of our algorithms is to the maintenance of strong components. This has been done for some of the earlier algorithms by previous authors. Pearce [19] and Pearce and Kelly [20] sketched how to extend their incremental topological ordering algorithm and that of Marchetti-Spaccamela et al. [17] to maintain strong components; HKMST showed in detail how to extend their algorithms. Here we describe how to extend ours.

Our strong-components algorithms maintain a representation of the ***condensation*** of the graph, which is the graph formed by contracting each strong component to a single vertex. We represent each vertex in the condensation by a unique ***canonical vertex*** in the corresponding component. We maintain the vertex sets of the components using a disjoint set data structure [28], which supports two operations:

FIND(x): Given a vertex x, return the canonical vertex of the set containing x.

LINK(x, y): Given two different canonical vertices x and y, unite the sets containing them into a single set whose canonical vertex is x. This operation destroys the old sets containing x and y.

Initially each vertex is a canonical vertex in its own singleton set. With an appropriate implementation of the set operations, the time for any sequence of intermixed LINK and FIND operations is $O(n \log n)$ plus $O(1)$ per operation [28]. In our strong-components algorithms the number of set operations is $O(1)$ per arc examined, so the time for the set operations does not increase the asymptotic time bounds for maintaining strong components.

We maintain the arcs in their original form and use FIND to transform them into arcs of the condensation: if (x, y) is an original arc, the corresponding arc of the condensation is $(\text{FIND}(x), \text{FIND}(y))$. Although the original graph contains no loops or multiple arcs, the condensation may contain such arcs. Such arcs can be ignored or deleted, since they do not affect the strong components or the possible topological orders of the components.

Each of our strong components algorithms runs the corresponding cycle-detection algorithm on the condensation. If an arc addition creates a new component, the algorithm does an extra depth-first search to find the vertices in the new component. Let G be an acyclic graph, and suppose the addition of arc (u, z) creates a cycle. Then the strong component containing (u, z) contains exactly the vertices on simple paths from z to u. We can find all such vertices by marking u and then doing a depth-first search forward from z. When retreating along an arc (x, y) during the search, we mark x if y is marked. When the search reaches u, we need not search recursively from u, but there is no harm in doing so. Once the search finishes, the marked vertices are those in the component. It is straightforward to verify by induction that this method correctly marks all vertices on simple paths from z to u. Equivalently, we can do a backward depth-first search from u.

5.1. Strong Components of Sparse Graphs. Our sparse strong-components algorithm maintains for each component a level, a set of arcs (x, y) such that x is in the component, and a set of arcs (x, y) such that y is in the component and the components containing x and y are on the same level. We store the level and the incident arc sets with the canonical vertex of the component. The levels are a pseudo topological numbering of the components: if (x, y) is an arc, then $k(\text{FIND}(x)) \leq k(\text{FIND}(y))$. Initially every vertex is in its own component, all components are on level 1, all the incident arc sets are empty, and all vertices are unmarked (not in a new component).

The algorithm for adding a new arc begins by running the sparse cycle-detection algorithm on the existing condensation, but it does not stop when it detects a cycle, it merely sets a bit indicating that a cycle exists. The backward search continues until it traverse Δ arcs or runs out of arcs to traverse; the forward search, if it occurs, continues until it traverses all arcs from vertices whose level increases. Once cycle detection is complete, the updated levels are a pseudo topological ordering of the current graph, which consists of the old condensation and the new arc; if the new arc (v, w) creates a new component, $\text{FIND}(v)$ and $\text{FIND}(w)$ are now on the same level, and a cycle has been detected. In this case, the algorithm does an extra depth-first search backward from $\text{FIND}(v)$, visiting only canonical vertices within the level of $\text{FIND}(v)$ and marking all canonical vertices in the new component. It then forms the new component by doing appropriate link operations.

Here are the details. The algorithm for adding a new arc (v, w) consists of five steps below. It sets the boolean variable *cycle*, initially FALSE, to TRUE when it detects a cycle. During the backward search in Step 2, it deletes loops and multiple arcs instead of traversing them. To facilitate the latter deletions, it uses a bit matrix M indexed by pairs of vertices. Initially M is identically zero.

Step 1 (test order): Set $u = \text{FIND}(v)$ and $z = \text{FIND}(w)$. If $k(u) < k(z)$, go to Step 4 (the levels remain a pseudo topological numbering).

Step 2 (search backward): Using the incoming arc sets, search backward from u, visiting only canonical vertices on the same level as u. Given a candidate arc (x, y)

for traversal, if $\textsc{Find}(x) = \textsc{Find}(y)$ or $M(\textsc{Find}(x), \textsc{Find}(y)) = 1$, delete (x, y) from $out(x)$ and from $in(y)$; otherwise, traverse (x, y) as follows: if $\textsc{Find}(x) = z$ then set $cycle = \textsc{true}$; otherwise, if $\textsc{Find}(x)$ is unvisited, mark x visited and add all arcs in $in(\textsc{Find}(x))$ to those to be traversed. Continue the search until at least Δ arcs are traversed or no backward arcs remain to be traversed. Reset all 1 entries in M to 0. Let B be the set of visited canonical vertices. If the search traverses fewer than Δ arcs and $k(z) = k(u)$, go to Step 4 (no forward search is needed). If the search traverses fewer than Δ arcs and $k(z) < k(u)$, set $k(z) = k(u)$. If the search traverses at least Δ arcs, set $k(z) = k(u) + 1$ and $B = \{u\}$. In either of the last two cases (those in which $k(z)$ increases), set $in(z) = \{\}$ and continue to Step 3.

Step 3 (search forward): Using the outgoing arc sets, search forward from z, following outgoing arcs only from canonical vertices whose level increases. The forward search updates the incoming arc sets as vertex levels increase. Specifically, when traversing a forward arc (x, y), if $\textsc{Find}(y) \in B$, set $cycle = \textsc{true}$. If $k(\textsc{Find}(y)) = k(z)$, add (x, y) to $in(\textsc{Find}(y))$. If $k(\textsc{Find}(y)) < k(z)$, set $k(\textsc{Find}(y)) = k(z)$, set $in(\textsc{Find}(y)) = \{(x, y)\}$, and add all arcs in $out(\textsc{Find}(y))$ to those to be traversed.

Step 4 (form component): If $cycle = \textsc{false}$ go to Step 5. Otherwise, proceed as follows. Set $cycle = \textsc{false}$. Mark z. Using the incoming arc sets, do a backward depth-first search from u, visiting only canonical vertices on the same level as u. When traversing a backward arc (x, y), if $\textsc{Find}(x)$ is marked, mark $\textsc{Find}(y)$. Otherwise, if $\textsc{Find}(x)$ is unvisited, visit $\textsc{Find}(x)$ and recursively search backward from $\textsc{Find}(x)$; once the recursive search finishes, if $\textsc{Find}(x)$ is marked, mark $\textsc{Find}(y)$. Once the search from u finishes, for each marked vertex $x \neq z$, do $\textsc{Link}(z, x)$, set $out(z) = out(z) \cup out(x)$, set $in(z) = in(z) \cup in(x)$, and unmark x. Finally, unmark z.

Step 5 (add arc): Add (v, w) to $out(u)$. If $k(u) = k(z)$, add (v, w) to $in(z)$.

In the proofs to follow we denote levels just before and just after the insertion of an arc (v, w) by unprimed and primed values, respectively.

THEOREM 5.1. *The sparse strong-components algorithm is correct. That is, it correctly maintains the strong components, all the data structures, and the following invariant on the levels: if (x, y) is an arc, then $k(\textsc{Find}(x)) \leq k(\textsc{Find}(y))$.*

Proof. The proof is by induction on the number of arc insertions. Initially all the data structures are correct. It is straightforward to verify that the algorithm correctly maintains them, assuming that it correctly maintains the strong components and the desired invariant on levels. Suppose the strong components are correct and the invariant holds before the insertion of an arc (v, w). The first three steps of the algorithm do the same thing as the first three steps of the unextended algorithm, except that they operate on the condensation instead of the original graph, and they do not stop when a cycle is detected but merely set $cycle = \textsc{true}$. It follows that if adding (v, w) does not create a now component, then after the addition the components are correct and the invariant holds.

Suppose on the other hand that adding (v, w) does create a new component. By the proof of Theorem 2.1, the algorithm will definitely set $cycle = \textsc{true}$. If a forward search does not occur, no levels change and $k(u) = k(z)$. If a forward search does occur, then z increases in level, and all canonical vertices reachable from z, including

u, have level at least that of z once the forward search finishes. It follows that at the beginning of Step 4, $k(\textsc{Find}(x)) \leq k(\textsc{Find}(y))$ for every original arc (x, y), and $k(u) = k(z)$. This means that the new component formed by the addition of (v, w) contains only canonical vertices on the same level as u and z, and the search in Step 4 will correctly find the vertices in the new component and correctly update the data structures. □

To bound the running time of the algorithm, we need to prove an analogue of Lemma 2.2. This requires some definitions. We call an arc (x, y) ***live*** if x and y are in different strong components and ***dead*** otherwise. A newly inserted arc that forms a new component is dead immediately. The ***level*** of a live arc (x, y) is $k(\textsc{Find}(x))$. The level of a dead arc is its highest level when it was live; an arc that was never live has no level. We identify each component with its vertex set; an arc insertion either does not change the components or combines two or more components into one. A component is ***live*** if it corresponds to a vertex of the current condensation and ***dead*** otherwise. The ***level*** of a live component is the level of its canonical vertex; the level of a dead component is its highest level when it was live. A vertex and a component are ***related*** if there is a path that contains the vertex and a vertex in the component. The number of components, live and dead, is at most $2n - 1$.

LEMMA 5.2. *No vertex level exceeds* $\min\left\{m^{1/2}, 2n^{2/3}\right\} + 1$ *in the sparse strong-components algorithm.*

Proof. We claim that for any level $k > 1$ and any level $j < k$, any canonical vertex of level k is related to at least Δ arcs of level j and at least $\sqrt{\Delta}$ components of level j. We prove the claim by induction on the number of arc insertions. The claim holds vacuously before the first insertion. Suppose it holds before the insertion of an arc (v, w). Let $u = \textsc{Find}(v)$ and $z = \textsc{Find}(w)$ before the insertion. A vertex is reachable from z after the insertion if and only if it is reachable from z before the insertion. The insertion increases the level only of z and possibly of some vertices and components reachable from z. It follows that the claim holds after the insertion for any canonical vertex not reachable from z.

Consider a vertex y that is reachable from z and is canonical after the insertion. Since levels are a pseudo topological numbering of the components, $k'(y) \geq k'(z)$. For j such that $k'(z) \leq j < k'(y)$, y is related to at least Δ arcs of level j and $\sqrt{\Delta}$ components of level j before the insertion. None of these arcs or components changes level as a result of the insertion, so the claim holds after the insertion for y and level j. Since any arc or component of level less than $k'(z)$ that is related to z is also related to y, the claim holds for y after the insertion if it holds for z.

After the insertion, z is reachable from u. Also, $k(u) \leq k'(z) \leq k(u) + 1$. The claim holds for u before the insertion. Let (x, y) be an arc of level less than $k(u)$ that is related to u before the insertion. If x is reachable from z, (x, y) will be dead after the insertion and hence its level will not change. Neither does its level change if x is not reachable from z. Arc (x, y) is related to z after the insertion. Consider a component of level less than $k(u)$ that is related to u before the insertion. If the component is reachable from z, it is dead after the insertion of (v, w) and hence does not change level; if it is not reachable from z, it also does not change level. After the insertion, the component is related to z. It follows that the claim holds for z and any level $j < k(u)$.

One case remains: $j = k(u) < k'(z) = k(u) + 1$. For the level of z to increase to

$k(u)+1$, the backward search must traverse at least Δ arcs of level $k(u)$ before the insertion, each of which is related to z and on level $k(u)$ after the insertion. The ends of these arcs are in at least $\sqrt{\Delta}$ components of level $k(u)$, each of which is related to z and on level $k(u)$ after the insertion. Thus the claim holds for z and level $k(u)$ after the insertion. This completes the proof of the claim.

The claim implies that for every level other than the maximum, there are at least Δ different arcs and $\sqrt{\Delta}$ different components. Since there are only m arcs and at most $2n-1$ components, the maximum level is at most $\min\left\{m/\Delta, 2n/\sqrt{\Delta}\right\}+1$. The lemma follows. □

THEOREM 5.3. *The sparse strong-components algorithm handles m arc insertions in $O(\min\left\{m^{1/2}, n^{2/3}\right\} m)$ time.*

Proof. The proof is like the proof of Theorem 2.3, using Lemma 5.2. The only new issue is that we must bound the time spent doing the search in Step 4. Each arc traversed in this search was either traversed in Step 2 or is an arc out of a vertex whose level increases. The number of arc traversals of the former kind is at most Δ per arc addition; the number of arc traversals of the latter kind is at most m times the maximum level. The maximum level is $O(\Delta)$ by Lemma 5.2. The time per arc traversal is $O(1)$, plus $O(n \log n)$ time over all arc examinations spent doing disjoint-set operations. □

The space required by the extended algorithm is $O(n^2)$, since the bit matrix M requires $O(n^2)$ space (or less if bits are packed into words). If we store M in a hash table, the space becomes $O(m)$ but the algorithm becomes randomized. By using a three-level data structure [29] to store M we can reduce the space to $O(n^{4/3}+m)$ without using randomization. We obtain a simpler algorithm with a time bound of $O(m^{3/2})$ by eliminating the deletion of multiple arcs, thus avoiding the need for M, and letting $\Delta = m^{1/2}$. If we run this simpler algorithm until $m > n^{4/3}$, then start over with all vertices on level one and indexed in topological order and run the more-complicated algorithm with M stored in a three-level data structure, we obtain a deterministic algorithm running in $O(\min\left\{m^{1/2}, n^{2/3}\right\} m)$ time and $O(m)$ space.

5.2. Strong Components of Dense Graphs. Our dense strong-components algorithm does two searches per arc addition, the first to find the new component if any, the second to update levels, bounds, and counts. The levels, bounds, counts, and arc heaps are of components, not vertices. We store these values with the canonical vertices of the components. Initially each vertex is in its own component, all levels are one, all bounds and counts are zero, and all heaps are empty. The algorithm deletes arcs with both ends in the same component, as well as the second and subsequent arcs between the same pair of components. As in the sparse extension, to do the latter it uses a bit matrix M indexed by pairs of vertices, initially identically zero. It also marks vertices found to be in a new component; initially all vertices are unmarked.

To insert an arc (v, w), let $u = \text{FIND}(x)$ and $z = \text{FIND}(y)$. If $k(u) < k(z)$, add (v, w) to $out(u)$ with priority $k(z)$. If $k(u) \geq k(z)$ and $u \neq z$, do Steps 1–4 below. (If $u = z$ do nothing.)

Step 1 (find component): Set $k(z) = k(u)+1$ and set $A = \{(v, w)\}$. Do a depth-first search forward from z, visiting only canonical vertices of level less than $k(z)$. To do the search, remove arcs (x, y) from $out(z)$ in non-decreasing order by priority until $out(z)$ is empty or the minimum priority of an arc in $out(z)$ is at least $k(z)$.

Given an arc (x, y), proceed as follows. Add (x, y) to A. If $\textsc{Find}(y) = u$, mark u. If $k(\textsc{Find}(y)) < k(z)$, set $k(\textsc{Find}(y)) = k(z)$ and search forward recursively from $\textsc{Find}(y)$; once the search finishes, if $\textsc{Find}(y)$ is marked, mark $\textsc{Find}(x)$.

Step 2 (form component): If z is marked, unite the components containing the marked canonical vertices into a single new component whose canonical vertex is z by doing appropriate Link operations. Form the new arc heap of z by melding the heaps of the marked vertices, including z. Unmark all marked vertices.

Step 3 (update levels, bounds, and counts): Repeat the following until A is empty:

Delete some arc (x, y) from A. If $\textsc{Find}(x) \neq \textsc{Find}(y)$ and $M(\textsc{Find}(x), \textsc{Find}(y)) = 0$, proceed as follows. Set $M(\textsc{Find}(x), \textsc{Find}(y)) = 1$. If $k(\textsc{Find}(x)) \geq k(\textsc{Find}(y))$, increase $k(\textsc{Find}(y))$ to $k(\textsc{Find}(x)) + 1$; if not, set $i = \lfloor \lg(k(\textsc{Find}(y)) - k(\textsc{Find}(x))) \rfloor$, add one to $c_i(\textsc{Find}(y))$, and if the counter reaches threshold $c_i(\textsc{Find}(y)) = 3 * 2^{i+1}$, set $c_i(\textsc{Find}(y)) = 0$, set $k(\textsc{Find}(y)) = \max\left\{k(\textsc{Find}(y)), b_i(\textsc{Find}(y)) + 3 * 2^i\right\}$, and set $b_i(\textsc{Find}(y)) = k(\textsc{Find}(y)) - 2^{i+1}$. Delete from $\mathit{out}(\textsc{Find}(y))$ each arc with priority at most $k(\textsc{Find}(y))$ and add it to A. Add (x, y) to $\mathit{out}(\textsc{Find}(x))$ with priority $k(\textsc{Find}(y))$.

Step 4 (reset M): Reset each 1 in M to 0.

Theorem 5.4. *The dense strong-components algorithm correctly maintains both strong components and the inequality $k(v) \leq \mathit{size}(v)$ for every vertex v.*

Proof. The proof is by induction on the number of arc insertions. The theorem holds initially. Suppose it holds before the insertion of an arc (v, w). Let $u = \textsc{Find}(v)$ and $z = \textsc{Find}(w)$ before the insertion, and consider vertex levels just before the insertion. If $u = z$ or $k(u) < k(z)$, the theorem holds after the insertion. Suppose $u \neq z$ and $k(u) \geq k(z)$, so that Steps 1–4 are executed. Step 1 visits all vertices of level less than $k(u) + 1$ reachable from z and increases their level to $k(u) + 1$. Since z is reachable from u after the insertion of (v, w), all such vertices have size at least $k(u) + 1$ after the insertion. Thus such increases in level maintain the inequality between levels and sizes. If the insertion of (v, w) creates a new component, the vertices in the component are exactly those on paths from z to u, all of which must have level at most $k(u)$ before the insertion. Thus Step 1 will visit and mark all such vertices, including u, and Step 2 will correctly form the new component. Step 3 updates levels, bounds, and counts exactly as in the unextended algorithm except that it operates on components, not vertices, and it traverses no arcs with both ends in the same component and at most one arc between any pair of components. The proof of Lemma 3.2 extends to show that Step 3 maintains the inequality between levels and sizes. Thus the theorem holds after the insertion. □

Theorem 5.5. *The dense strong-components algorithm runs in $O(n^2 \log n)$ total time.*

Proof. The proof of Lemma 3.4 extends to show that the extended dense algorithm does $O(n^2 \log n)$ arc traversals, from which the theorem follows. □

The space required by the extended dense algorithm is $O(n^2)$, or $O(n \log n + m)$ if the heaps and the matrix M are stored in hash tables.

5.3. Topological Order. It is straightforward to apply the ideas from Sections 2 and 4 to extend the spare and dense strong-components algorithms to maintain a weak topological numbering of the components and/or a list of the components in a topological order. We leave this extension as an exercise.

6. Concluding Remarks. We have presented two algorithms for incremental cycle detection and related problems, one for sparse graphs and one for dense graphs. Their total running times are $O(\min\{m^{1/2}, n^{2/3}\}\, m)$ and $O(n^2 \log n)$, respectively. The sparse algorithm is faster for graphs whose density m/n is $o(n^{1/3} \log n)$; the dense algorithm is faster for graphs of density $\omega(n^{1/3} \log n)$. The $O(n^{2/3}m)$ bound of the sparse algorithm is best only for graphs with density in the sliver from $\omega(n^{1/3})$ to $o(n^{1/3} \log n)$. The HKMST paper gives a lower bound of $\Omega(nm^{1/2})$ for algorithms that maintain an explicit list of the vertices in a topological order and do vertex updates only within the so-called "affected region," the set of vertices that are definitely out of order when a new arc is added. Unlike previous algorithms, our algorithms do not do updates completely within the affected region, yet they do not beat the HKMST lower bound, and we have no reason to believe it can be beaten. On the other hand, for graphs of intermediate density our bounds are far from $O(nm^{1/2})$, and perhaps improvements coming closer to this bound are possible.

Another interesting research direction is to investigate whether batch arc additions can be handled faster than single arc additions (other than by reverting to a static algorithm if the batch is large enough). See [4, 23]. One may also ask whether arc deletions, instead of or in addition to insertions, can be handled. Our cycle-detection and topological-ordering algorithms remain correct if arcs can be deleted as well as inserted, but the time bounds are no longer valid, and we have no interesting bounds. Maintaining strong components as arcs are deleted, or as arcs are inserted and deleted, is an even more challenging problem. See [24] and the references contained therein.

Acknowledgment. The fourth author thanks Don Knuth who, during a presentation of a previous version of the sparse cycle-detection algorithm, asked whether the algorithm really needed to use indices in addition to levels. The simplified algorithm presented in Section 2 demonstrates that the answer is no.

REFERENCES

[1] Deepak Ajwani and Tobias Friedrich, *Average-case analysis of online topological ordering*, in ISAAC, Takeshi Tokuyama, ed., vol. 4835 of Lecture Notes in Computer Science, Springer, 2007, pp. 464–475.

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

[3] ———, *An $O(n^{2.75})$ algorithm for online topological ordering*, ACM Transactions on Algorithms, 4 (2008), pp. 39:1–39:14.

[4] Bowen Alpern, Roger Hoover, Barry K. Rosen, Peter F. Sweeney, and F. Kenneth Zadeck, *Incremental evaluation of computational circuits*, in Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, 1990, pp. 32–42.

[5] Michael A. Bender, Richard Cole, Erik D. Demaine, Martin Farach-Colton, and Jack Zito, *Two simplified algorithms for maintaining order in a list*, in Proceedings of the European Symposium on Algorithms, 2002, pp. 152–164.

[6] Michael A. Bender, Jeremy T. Fineman, and Seth Gilbert, *A new approach to incremental topological ordering*, in Proceedings of the ACM-SIAM Symposium on Discrete Algorithms, January 2009.

[7] Paul F. Dietz and Daniel D. Sleator, *Two algorithms for maintaining order in a list*, in Proceedings of the ACM Symposium on the Theory of Computing, May 1987, pp. 365–372.
[8] Bernhard Haeupler, Telikepalli Kavitha, Rogers Mathew, Siddhartha Sen, and Robert E. Tarjan, *Faster algorithms for incremental topological ordering*, in Proceedings of the 35th International Colloquium on Automata, Languages, and Programming, July 2008, pp. 421–433.
[9] Bernhard Haeupler, Telikepalli Kavitha, Roger Mathew, Siddhartha Sen, and Robert E. Tarjan, *Incremental cycle detection, toplogical ordering, and strong component maintenance*, ACM Transactions on Algorithms, 8 (2012), pp. 3:1–3:33.
[10] Frank Harary, Robert Z. Norman, and Dorwin Cartwright, *Structural Models: An Introduction to the Theory of Directed Graphs*, John Wiley & Sons, 1965.
[11] A.B. Kahn, *Topological sorting of large networks*, Communications of the ACM, 5 (1962), pp. 558–562.
[12] Irit Katriel and Hans L. Bodlaender, *Online topological ordering*, in Proceedings of the 16th ACM-SIAM Symposium on Discrete Algorithms, Vancouver, British Columbia, Canada, January 2005, pp. 443–450.
[13] ———, *Online topological ordering*, ACM Transactions on Algorithms, 2 (2006), pp. 364–379.
[14] Donald E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Addison Wesley, Reading, MA, 2nd ed., 1973.
[15] Donald E. Knuth and Jayme L. Szwarcfiter, *A structured program to generate all topological sorting arrangements*, Information Processing Letters, 2 (1974), pp. 153–157.
[16] Hsiao-Fei Liu and Kun-Mao Chao, *A tight analysis of the katriel–bodlaender algorithm for online topological ordering*, Theoretical Computer Science, 389 (2007), pp. 182–189.
[17] Alberto Marchetti-Spaccamela, Umberto Nanni, and Hans Rohnert, *Maintaining a topological order under edge insertions*, Information Processing Letters, 59 (1996), pp. 53–58.
[18] D.J. Pearce, P.H.J. 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, Sept. 2003, pp. 3–12.
[19] David J. Pearce, *Some Directed Graph Problems and Their Application to Pointer Analysis*, PhD thesis, 2005.
[20] David J. Pearce and Paul H. J. Kelly, *Online algorithms for topological order and strongly connected components*, tech. report, Imperial College, London, 2003.
[21] ———, *A dynamic algorithm for topologically sorting directed acyclic graphs*, in Proceedings of the 3rd International Workshop on Efficient Experimental Algorithms, vol. 3059 of Lecture Notes in Computer Science, Springer, 2004, pp. 383–398.
[22] ———, *A dynamic topological sort algorithm for directed acyclic graphs*, Journal of Experimental Algorithms, 11 (2006).
[23] ———, *A batch algorithm for maintaining a topological order*, in Proceedings of the Thirty-Third Australasian Conference on Computer Science, ACSC '10, 2010, pp. 79–88.
[24] Liam Roditty and Uri Zwick, *Improved dynamic reachability algorithms for directed graphs*, SIAM J. Comput., 37 (2008), pp. 1455–1471.
[25] E. Szpilrajn, *Sur l'extension de l'ordre partiel*, Fundamenta Mathematicae, 16 (1930), pp. 386–389.
[26] Robert E. Tarjan, *Depth-first search and linear graph algorithms*, SIAM J. Computing, 1 (1972), pp. 146–160.
[27] ———, *Finding dominators in directed graphs*, SIAM J. Computing, 3 (1974), pp. 62–89.
[28] ———, *Efficiency of a good but not linear set union algorithm*, J. Assoc. Comput. Mach., 22 (1975), pp. 215–225.
[29] Robert E. Tarjan and Andrew Chi-Chih Yao, *Storing a sparse table*, Communications of the ACM, 22 (1979), pp. 606–611.