Faster Algorithms for Incremental Topological Ordering*

Bernhard Haeupler[1], Telikepalli Kavitha[2], Rogers Mathew[2], Siddhartha Sen[1], and Robert E. Tarjan[1,3]

[1] Princeton University, Princeton NJ 08544
{haeupler, sssix, ret}@cs.princeton.edu
[2] Indian Institute of Science, Bangalore, India.
{kavitha, rogers}@csa.iisc.ernet.in
[3] HP Laboratories, Palo Alto CA 94304

Abstract. We present two online algorithms for maintaining a topological order of a directed acyclic graph as arcs are added, and detecting a cycle when one is created. Our first algorithm takes $O(m^{1/2})$ amortized time per arc and our second algorithm takes $O(n^{2.5}/m)$ amortized time per arc, where n is the number of vertices and m is the total number of arcs. For sparse graphs, our $O(m^{1/2})$ bound improves the best previous bound by a factor of $\log n$ and is tight to within a constant factor for a natural class of algorithms that includes all the existing ones. Our main insight is that the two-way search method of previous algorithms does not require an ordered search, but can be more general, allowing us to avoid the use of heaps (priority queues). Instead, the deterministic version of our algorithm uses (approximate) median-finding; the randomized version of our algorithm uses uniform random sampling. For dense graphs, our $O(n^{2.5}/m)$ bound improves the best previously published bound by a factor of $n^{1/4}$ and a recent bound obtained independently of our work by a factor of $\log n$. Our main insight is that graph search is wasteful when the graph is dense and can be avoided by searching the topological order space instead. Our algorithms extend to the maintenance of strong components, in the same asymptotic time bounds.

1 Introduction

We consider two related problems on dynamic directed graphs: cycle detection and maintaining a topological order. These problems are closely connected, since a directed graph has a topological order if and only if it is acyclic. We present two algorithms for maintaining a topological order that asymptotically improve the best known time bounds for both sparse and dense graphs.

A *topological order* of a directed graph is a total order of the vertices such that for every arc (v, w), $v < w$. A directed graph has a topological order (and generally more than one) if and only if it is acyclic [9]. Given a fixed n-vertex, m-arc graph, a

* This paper combines the best results of [12] and [7]. The former gives two incremental topological ordering algorithms, with amortized time bounds per arc addition of $O(m^{1/2} + (n \log n)/m^{1/2})$ and $O(n^{2.5}/m)$. The latter, which was written with knowledge of the former, gives an algorithm with a bound of $O(m^{1/2})$.

topological order can be found in $O(n+m)$ time by either of two algorithms: repeated deletion of sources [13, 14] or depth-first search [28]. The former method extends to the enumeration of all possible topological orderings.

In some applications, the graph is not fixed but changes over time. The *incremental topological ordering problem* is that of maintaining a topological order of a directed graph as arcs are added, stopping when addition of an arc creates a cycle. This problem arises in incremental circuit evaluation [2], pointer analysis [21], management of compilation dependencies [17, 19], and deadlock detection [3].

In considering the incremental topological ordering problem, we shall assume that the vertex set is fixed and specified initially, and that the arc set is initially empty. Our ideas easily extend to support vertex as well as arc additions, with a time of $O(1)$ per vertex addition. Our methods also allow arc deletions, since deletion of an arc cannot create a cycle nor invalidate the current topological order, but our time bounds are only valid if there are no arc deletions. Our algorithms also extend to the maintenance of strong components under arc additions, in the same asymptotic time bounds, but we omit the details here because of space constraints.

One could of course recompute a topological order after each arc addition, but this would require $O(n+m)$ time per arc. Our goal is to do better. The incremental cycle detection and topological ordering problems have received much attention, especially recently. Shmueli [27] applied depth-first search to test for cycles in a directed graph subject to arc additions and deletions. As a heuristic to improve its efficiency, his method maintains a topological ordering, although he did not describe it as such. For the topological ordering problem, Marchetti-Spaccamela et al. [18] gave a one-way search algorithm that takes $O(n)$ amortized time per arc addition. Alpern et al. [2] gave a two-way search algorithm that handles batched arc additions and has a good time bound in an incremental model of computation. Katriel and Bodlaender [11] showed that a variant of the algorithm of Alpern et al. takes $O(\min\{m^{1/2}\log n, m^{1/2}+(n^2\log n)/m\})$ amortized time per arc addition. Liu and Chao [15] tightened this analysis to $\Theta(m^{1/2}+n^{1/2}\log n)$ per arc addition. Pearce and Kelly [20] gave an algorithm that they claimed was fast in practice on sparse graphs, although it has an inferior asymptotic time bound. Kavitha and Mathew [12] improved the results of Liu and Chao by presenting another variant of the algorithm of Alpern et al. with an $O(m^{1/2}+(n\log n)/m^{1/2})$ amortized time bound per arc addition. Ajwani et al. [1] gave an algorithm with an amortized time per arc addition of $O(n^{2.75}/m)$, better for dense graphs than the bound of Kavitha and Mathew. Independent of our work, Liu and Chao [16] modified the algorithm of Ajwani et al. to obtain an $\tilde{O}(n^{2.5}/m)$ amortized time bound per arc addition.

We generalize the two-way search method introduced by Alpern et al. and used in later algorithms by observing that the method does not require an ordered search (used in all previous algorithms) to be correct. This allows us to refine the general method into one that avoids the use of heaps (priority queues), but instead uses either approximate median-finding or random selection, resulting in an $O(m^{1/2})$ amortized time bound per arc addition. The randomized version of our algorithm is especially simple.

For dense graphs, we observe that graph search itself is wasteful because it can do unnecessary arc traversals. By searching the current topological order instead, we obtain an $O(n^{2.5}/m)$ amortized time bound per arc addition. This algorithm is also

simple, although its analysis relies on a linear program bound developed by Ajwani et al.

The body of our paper comprises five sections. Section 2 describes the two-way search method, verifies its correctness, and analyzes its running time. Section 3 refines the method to yield an algorithm fast for sparse graphs. Section 4 describes an alternative approach that avoids graph search and yields an algorithm fast for dense graphs. Section 5 analyzes the running time of this algorithm. Section 6 examines lower bounds and other issues. In particular, we show in this section that on sparse graphs our $O(m^{1/2})$ amortized time bound is tight to within a constant factor for a natural class of algorithms that includes all the existing ones.

2 Topological Ordering via Two-Way Search

We develop our topological ordering algorithm through refinement of a general method that encompasses many of the older algorithms. By vertex order we mean the current topological order. We maintain the vertex order using a data structure such that testing the predicate "$x < y$" for two vertices x and y takes $O(1)$ time, as does deleting a vertex from the order and reinserting it just before or just after another vertex. The dynamic ordered list implementations of Dietz and Sleator [6] and Bender et al. [4] meet these requirements: their methods take $O(1)$ time worst-case for an order query or a deletion, and $O(1)$ time for an insertion, amortized or worst-case depending on the structure. For us an amortized bound suffices. In addition to a topological order, we maintain the outgoing and incoming arcs of each vertex. This allows two-way search. Initially the vertex order is arbitrary and all sets of outgoing and incoming arcs are empty.

To add an arc (v, w) to the graph, proceed as follows. Add (v, w) to the set of arcs out of v and to the set of arcs into w. If $v > w$, search forward from w and backward from v until finding either a cycle or a set of vertices whose reordering will restore topological order.

A vertex x is *forward* if there is a path from w to x of arcs traversed forward, *backward* if there is a path from x to v of arcs traversed backward. A vertex is *scanned* if it is forward and all its outgoing arcs have been traversed, or it is backward and all its incoming arcs have been traversed. To do the search, traverse arcs forward from forward vertices and backward from backward vertices until either a forward traversal reaches a backward vertex x or a backward traversal reaches a forward vertex x, in which case there is a cycle, or until there is a vertex s such that all forward vertices less than s and all backward vertices greater than s are scanned.

In the former case, stop and report a cycle consisting of a path from w to x traversed forward, followed by a path from x to v traversed backward, followed by (v, w). In the latter case, restore topological order as follows. Let X be the set of forward vertices less than s and Y the set of backward vertices greater than s. Find topological orders $\vec{X}$ and $\vec{Y}$ of the subgraphs induced by X and Y, respectively. Assume s is not forward; the case of s not backward is symmetric. Delete the vertices in $X \cup Y$ from the current vertex order and reinsert them just after s, in order $\vec{Y}$ followed by $\vec{X}$. (See Figure 1.)

Theorem 1. *The two-way search method is correct.*

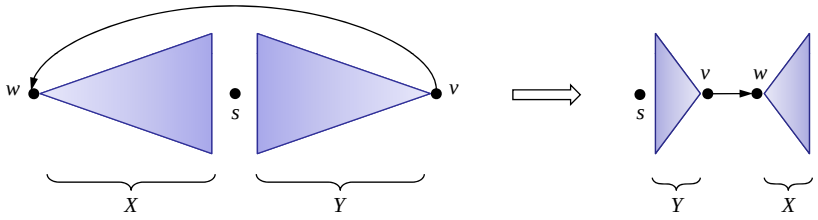

Fig. 1. Restoring topological order after two-way search. Since s is not forward, the vertices in Y are inserted (in topological order) just after s, followed by the vertices in X.

Proof. Omitted. See [7]. □

This method requires $O(1)$ time plus $O(1)$ time per arc traversed, plus any overhead needed to guide the search. To obtain a good time bound we need to minimize both the number of arcs traversed and the search overhead. In our discussion we shall assume that no cycle is created. Only one arc addition, the last one, can create a cycle; the last search takes $O(m)$ time plus overhead.

We need a way to charge the search time against graph changes caused by arc additions. To measure such changes, we count pairs of related graph elements, either vertex pairs, vertex-arc pairs, or arc-arc pairs: two such elements are *related* if they are on a common path. The number of related pairs is initially zero and never decreases. There are at most $\binom{n}{2} < n^2/2$ vertex-vertex pairs, nm vertex-arc pairs, and $\binom{m}{2} < m^2/2$ arc-arc pairs. For sparse graphs, the related arc-arc pairs are of most use to us.

We limit the search in three ways to make it more efficient. First, we restrict it to the *affected region*, the set of vertices between w and v. Specifically, only arcs (u, x) with $u < v$ are traversed forward, and only arcs (y, z) with $z > w$ are traversed backward. This suffices to attain an $O(n)$ amortized time bound per arc addition. The bound comes from a count of newly-related vertex-arc pairs: each arc (u, x) traversed forward is newly related to v, each arc (y, z) traversed backward is newly related to w. The algorithm of Marchetti-Spaccamela et al. [18] is the special case that does just a one-way search forward from w using $s = v$, with one refinement and one difference: it does a depth-first search, and it maintains the topological order as an explicit mapping between the vertices and the integers from 1 to n.

One-way search allows a more space-efficient graph representation, since we need only forward incidence sets, not backward ones. But two-way search has a better time bound if it is suitably limited. We make the search *balanced*: each traversal step is of two arcs concurrently, one forward and one backward. There are other balancing strategies [2, 11, 12], but this simple one suffices for us. Balancing by itself does not improve the time bound; we need a third restriction. We call an arc (u, x) traversed forward and an arc (y, z) traversed backward *compatible* if $u < z$. Compatibility implies that (u, x) and (y, z) are newly related. We make the search *compatible*: each traversal step is of two compatible arcs; the search stops when there is no pair of untraversed compatible arcs.

Theorem 2. *Compatible search is correct.*

Proof. Omitted. □

Lemma 1. *If the searches are compatible, the amortized number of arcs traversed during searches is $O(m^{1/2})$ per arc addition.*

Proof. We count related arc-arc pairs. Consider a compatible search of $2k$ arc traversals, k forward and k backward. Order the arcs (u, x) traversed forward in increasing order on u, breaking ties arbitrarily. Let (u, x) be the $\lceil k/2 \rceil^{th}$ arc in the order. Arc (u, x) and each arc following (u, x) has a compatible arc (y, z) traversed backward. Compatibility and the ordering of forward traversed arcs imply that $u < z$. Thus each such arc (y, z) is newly related to (u, x) and to each arc preceding (u, x), for a total of at least $(k/2)^2$ newly related pairs.

We divide searches into two kinds: those that traverse at most $m^{1/2}$ arcs and those that traverse more. Searches of the first kind satisfy the bound of the lemma. Let $2k_i$ be the number of arcs traversed during the i^{th} search of the second kind. Since $2k_i > m^{1/2}$ and $\sum_i (k_i/2)^2 < m^2/2$, $\sum_i k_i < 2\sum_i {k_i}^2/m^{1/2} = 8\sum_i (k_i/2)^2/m^{1/2} < 4m^{3/2}$. Thus there are $O(m^{1/2})$ arc traversals per arc addition. □

We still need a way to implement compatible search. The most straightforward is to make the search ordered: traverse arcs (u, x) forward in non-decreasing order on u and arcs (y, z) backward in non-increasing order on z. This requires two heaps (priority queues) to store unscanned forward and unscanned backward vertices. In essence this is the algorithm of Alpern et al. [2], although they use a different balancing strategy. The heap overhead is $O(\log n)$ per arc traversal, resulting in an amortized time bound of $O(m^{1/2} \log n)$ per arc addition. More-complicated balancing strategies lead to the improvements [11, 12, 15] in this bound for non-dense graphs mentioned in Section 1.

3 Compatible Search via a Soft Threshold

The running time of an ordered search can be reduced further, even for sparse graphs, by using a faster heap implementation, such as those of van Emde Boas [31, 30], Thorup [29], and Han and Thorup [8]. But we can do even better, avoiding the use of heaps entirely, by exploiting the flexibility of compatible search. We guess the value of the threshold s and traverse arcs forward only from vertices less than or equal to s and backward only from vertices greater than or equal to s. When we run out of unscanned vertices on one side or the other, we revise our guess of s. Since this method does not know the final value of s until it stops, it can do extra work, but with a careful choice of s this extra work only costs a constant factor.

In addition to the soft threshold s, the algorithm maintains two hard thresholds, l and h, such that all forward vertices less than l and all backward vertices greater than h are scanned. It maintains the invariant $l \leq s \leq h$. It also maintains partitions of the candidate forward vertices into $A \cup B$ and of the candidate backward vertices into $C \cup D$. Set B contains forward vertices temporarily bypassed because they are greater than s (but less than h); set D contains backward vertices temporarily bypassed because they are less than s (but greater than l). We call the vertices in $B \cup D$ *far*. The remaining candidate vertices, those in $A \cup C$, are *near*. The algorithm is as follows. Initialize l to

w, h to v, s to v (or w), A to $\{w\}$, C to $\{v\}$, and B, D, X, and Y to empty. Then repeat an applicable one of the following cases until $A \cup B$ or $C \cup D$ is empty (see Figure 2):

Case 1f. $A = \emptyset$: set $l = s$, $A = B$, $B = D = \emptyset$, and choose $s \in A$.
Case 1b. $C = \emptyset$: set $h = s$, $C = D$, $B = D = \emptyset$, and choose $s \in C$.

In the remaining cases, A and C are non-empty. Choose $u \in A$ and $z \in C$.

Case 2f. $u > s$: delete u from A; if $u < h$, insert u in B.
Case 2b. $z < s$: delete z from C; if $z > l$, insert z in D.

In the remaining cases $l \leq u \leq s \leq z \leq h$.

Case 3f. All arcs from u are traversed: move u from A to X.
Case 3b. All arcs to z are traversed: move z from C to Y.
Case 4. There are untraversed arcs (u, x) and (y, z): choose two such arcs and traverse them. If x is backward or y is forward, stop and report a cycle. If x is unreached and less than h, make it forward and add it to A. If y is unreached and greater than l, make it backward and add it to C.

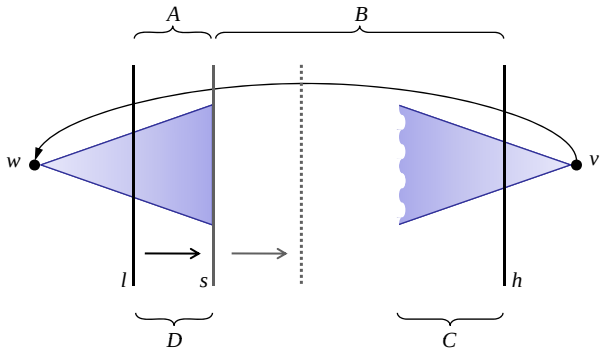

Fig. 2. Compatible search via a soft threshold. In this example A is empty, so Case 1f applies: l is moved to s and a new s is selected from the vertices in B (which becomes the new A).

If the search stops without detecting a cycle, set $s = h$ if $A \cup B$ is empty, $s = l$ otherwise. Delete from X all forward vertices no less than s and from Y all backward vertices no greater than s. Reorder the vertices as described in Section 2.

Theorem 3. *Compatible search with a soft threshold is correct.*

Proof. Omitted. See [7]. □

Lemma 2. *The running time of compatible search via a soft threshold is $O(1)$ plus $O(1)$ per arc traversed plus $O(1)$ for each time a vertex becomes near.*

Proof. Each case either traverses two arcs and adds at most two vertices to $A \cup C$, or permanently deletes a vertex from $A \cup C$, or moves a vertex from $A \cup C$ to $B \cup D$, or moves one or more vertices from $B \cup D$ to $A \cup C$. The number of times vertices are moved from $A \cup C$ to $B \cup D$ is at most the number of times vertices become near. □

The algorithm is correct for any choice of soft threshold, but only a careful choice makes it efficient. Repeated cycling of vertices between near and far is the remaining inefficiency. We choose the soft threshold to limit such cycling no matter how the search proceeds. A good deterministic choice is to let the soft threshold be the median or an approximate median of the appropriate set (A or C); an *ϵ-approximate median* of a totally ordered set of g elements is any element that is less than or equal to at least ϵg elements and greater than or equal to at least ϵg elements, for some constant $\epsilon > 0$. The (exact) median is a $1/2$-approximate median. Finding the median or an approximate median takes $O(g)$ time [5, 26]. An alternative is to choose the soft threshold uniformly at random from the appropriate set. This gives a very simple yet efficient randomized algorithm.

Lemma 3. *If each soft threshold is an ϵ-approximate median of the set from which it is chosen, then the number of times a vertex becomes near is $O(1)$ plus $O(1)$ per arc traversed. If each soft threshold is chosen uniformly at random, then the expected number of times a vertex becomes near is $O(1)$ plus $O(1)$ per arc traversed.*

Proof. The value of l never decreases as the algorithm proceeds; the value of h never increases. Let k be the number of arcs traversed. Suppose each soft threshold is an ϵ-approximate median. The first time a vertex is reached, it becomes near. Each subsequent time it becomes near, it is one of a set of g vertices that become near, as a result of being moved from B to A or from D to C. The two cases are symmetric; consider the former. No matter what happens later, at least ϵg vertices have become near for the last time. Just after s is changed, at least ϵg vertices in A are no less than s, and at least ϵg vertices in A are no greater than s. Just before the next time s changes, $l = s$ or $h = s$. In the former case, all vertices no greater than s can never again become near; in the latter case, all vertices no less than s can never again become near. We charge the group of g newly near vertices to the vertices that become near for the last time. The total number of times vertices can become near is at most $(2 + k)/\epsilon$: there are at most $2 + k$ forward and backward vertices and at most $1/\epsilon$ times a vertex can become near per forward or backward vertex.

Essentially the same argument applies if the soft threshold is chosen uniformly at random. If a set of g vertices becomes near, the expected number that become near for the last time is at least $\sum_{1 \le i \le g/2}(2i)/g = (g/2 + 1)/2 > g/4$ if g is even, at least $\lceil g/2 \rceil + \sum_{1 \le i < \lfloor g/2 \rfloor}(2i)/g = \lceil g/2 \rceil^2/g > g/4$ if g is odd. The total expected number of times vertices can become near is at most $4(2 + k)$. □

Theorem 4. *The amortized time for incremental topological ordering via compatible search is $O(m^{1/2})$ per arc addition, worst-case if each soft threshold is an ϵ-approximate median of the set from which it is chosen, expected if each soft threshold is chosen uniformly at random.*

Proof. Immediate from Lemmas 1–3. □

4 The Dense Case: Topological Search

The compatible search method described in Section 3 is efficient for sparse graphs. In dense graphs, graph search becomes wasteful because it can do unnecessary arc traversals, in particular of arcs that end at vertices beyond the stopping threshold. One way to reduce the overhead of graph search is to sort the incident arc lists by end vertex. Unfortunately, keeping the arc lists sorted seems to require more than $O(m^{3/2})$ time, giving no actual improvement. The $O(n^{2.75})$-time algorithm of Ajwani et al. uses this idea but keeps the arc lists partially sorted, trading off search time against arc list reordering time.

We do better for dense graphs by avoiding graph search and instead searching the topological order. We change the representations of both the graph and the vertex order. For the former we use a matrix $M : M(v, w) = 1$ if (v, w) is an arc, 0 otherwise. For the latter we use an explicit 1-1 mapping $I : V \to \{1, \ldots, n\}$; we also maintain I^{-1}. For a given new arc (v, w) with $I(v) > I(w)$, we visit the vertices in topological order forward from w and backward from v, accessing consecutive entries of I^{-1} until finding either a cycle or a set of vertices whose reordering will restore topological order. This is a form of ordered two-way search in which the ordering comes for free because we search the order, not the graph. The difficulty lies in finding the set of vertices that need to be reordered, because we can no longer depend on arc traversals to find paths.

We explain our algorithm in detail now. We maintain the set of forward vertices F, or those that can be reached from w, and backward vertices R, or those that can reach v, as deques [13] (double-ended queues). To add an arc (v, w) with $I(w) = i$ and $I(v) = j$, proceed as follows. Set $M(v, w) = 1$. If $v > w$, initialize F to $\{w\}$ and R to $\{v\}$ and do an ordered two-way search forward from i and backward from j in $I^{-1}[i..j]$. For each index k visited by the forward search, determine if the vertex $x = I^{-1}[k]$ is forward by querying $M(f, x)$ for each $f \in F$. If any of the queries evaluates to 1, add x to the back of F. The backward search is symmetric. Stop the search when an index t is reached such that all vertices in $I^{-1}[i..t]$ have been visited by the forward search and all vertices in $I^{-1}[t..j]$ have been visited by the backward search. (See Figure 3.)

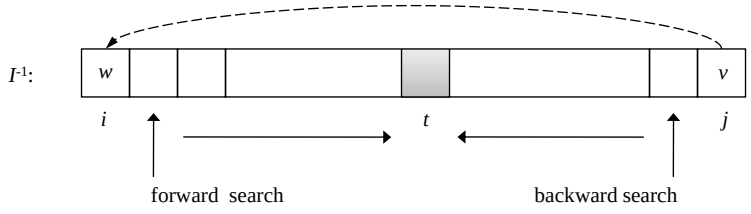

Fig. 3. The forward and backward search meet at index t.

As in our sparse algorithm, we limit the search in several ways to make it more efficient. First, we restrict it to the affected region $I^{-1}[i..j]$. We also make the search balanced: we alternate searching forward and backward to balance the size of F and R.

The forward search runs until it adds a vertex to F, then the backward search runs until it adds a vertex to R, and so on.

At the end of the search phase, we can determine if (v, w) creates a cycle using the following lemma:

Lemma 4. *The new arc (v, w) creates a cycle if and only if $I^{-1}[t] \in F \cap R$ or there is an arc (x, y) such that $x \in F$ and $y \in R$.*

Proof. Omitted. See [12]. □

We still need to reorder the vertices in F and R to restore topological order. We can use the method described in Section 2, but this breaks the 1-1 mapping between the vertices and $\{1, \ldots, n\}$. We take a much simpler approach: we reuse the indices of the vertices in $F \cup R$ (and possibly some other indices we have not seen yet) to reorder the vertices. Begin by deleting all the vertices of $F \cup R$ from their current locations in I^{-1}. The vertices in F must find new indices in $I^{-1}[t..j]$ and the vertices in R must find new indices in $I^{-1}[i..(t-1)]$. Extend the forward search to $I^{-1}[t..j]$; the backward search is extended symmetrically to $I^{-1}[i..(t-1)]$. For each index $k \in \{t, \ldots, j\}$, delete the vertex at the front of F and place it in $I^{-1}[k]$ if $I^{-1}[k]$ is empty. Otherwise if the vertex $x = I^{-1}[k]$ is forward, remove x from its current location and add it to the end of F and place the vertex at the front of F into $I^{-1}[k]$.

At the beginning of the reordering phase the size of F and R are equal to within 1 because they are balanced during the search phase. Consider the forward search; the backward search is symmetric. If the forward search started first during the search phase, then F has at most 1 more vertex than R, the last vertex being at index t itself. It follows that the number of empty indices in $I^{-1}[t..j]$ equals the size of F after the vertices in $F \cup R$ are deleted from I^{-1}. Each time a forward vertex is found during the reordering phase it is added to F and its location in I^{-1} is filled by an existing vertex of F. Each empty index is filled by a vertex from F. Thus the number of empty indices in $I^{-1}[k..j]$ for $k \geq t$ always equals the size of F when the forward search is at k. When $k = j$ there are no empty indices left and all vertices in F have been placed in $I^{-1}[t..j]$.

It is easy to see that the reordering scheme above preserves the relative order of the vertices in F and the relative order of the vertices in R. The reordering is effectively a cyclic permutation of the forward and backward vertices in the affected region. Vertices that are not forward or backward, and all vertices outside the affected region, are unaffected by the algorithm. Combining the above argument with Lemma 4 and Theorem 1, we have:

Theorem 5. *Topological search is correct.*

5 Bounding the Running Time

To bound the running time of the algorithm, observe that its work comes in two forms: searching the topological order between i and j and maintaining the sets F and R, and checking for a cycle prior to the reordering phase. Since the two-way search is ordered, the latter time can be bounded using a count of related vertex pairs (all pairs in $F \times R$ prior to the reordering phase are newly related).

Lemma 5. *The time required to check for cycles over all arc additions is* $O(n^2)$.

It is instructive to observe that balanced search is not required for Lemma 5. Balanced search is used to draw an equivalence between the total movement of vertices during reordering and the work performed by the algorithm to maintain the sets F and R. Let I_e be the topological order before adding arc $e = (v, w)$ and let I'_e be the topological order after adding e. Balanced search allows us to show the following:

Lemma 6. *The time required to maintain* F *and* R *for a new arc* e *is* $\sum_{x \in V} |I_e(x) - I'_e(x)|$.

Proof. Consider the forward search; the backward search is symmetric. Any vertex $x \in F$ belongs to the set F while the forward search moves from $I_e(x)$ to $I'_e(x)$ in I^{-1}. During this time, each non-empty index k causes the algorithm to check if the vertex $I^{-1}[k]$ is adjacent to x. Since x is involved in only one such query per index, the total work attributable to x is $|I_e(x) - I'_e(x)|$. To see why x does not do more than this amount of work, it suffices to observe that F and R are balanced prior to reordering and that $I'_e(x)$ must lie in $\{t, \ldots, j\}$.

Now, we use a result from [1] to bound the total movement of vertices, which in turn bounds the total running time of our algorithm.

Lemma 7. $\sum_{x \in V} |I_e(x) - I'_e(x)|$ *over all arcs* e *is* $O(n^{2.5})$.

Proof. The reordering phase of our algorithm performs a cyclic permutation of the vertices inserted into $F \cup R$. We can view this permutation as a sequence of swaps between pairs of vertices (x, y) where $x \in R$ and $y \in F$. Prior to a swap $I(x) > I(y)$. If a pair (x, y) is swapped during a permutation, then it is never swapped again in any future permutation because x and y are newly related by the path created from x to v followed by the arc (v, w) followed by the path from w to y. Let $d(x, y) = I(x) - I(y)$, the difference between x and y when they are swapped as a result of a permutation. By the previous argument $d(x, y)$ is uniquely defined. Ajwani et al. [1] used a linear program to show that $\sum d(x, y) = O(n^{2.5})$, where the summation is over all pairs (x, y) that get swapped during some permutation. Thus it suffices to decompose a permutation into a sequence of swaps to prove this lemma.

Let F_e be the set of vertices added to F as a result of processing arc e; define R_e analogously. (F_e and R_e contain vertices inserted during both the search and reordering phases of the algorithm.) Let $F_e = \{w_0, w_1, \ldots, w_p\}$, where $w_0 = w$ and $O(w_0) < \ldots < O(w_p)$, and let $R_e = \{v_0, v_1, \ldots, v_q\}$, where $v_0 = v$ and $O(v_0) > \ldots > O(v_q)$. We decompose the permutation of $F_e \cup R_e$ as follows. For each vertex $x \in R_e$ starting with v_q and in increasing order, swap x with the vertices in F_e in decreasing order, starting with the first vertex $w_r \in F_e$ that is less than x. That is, swap x successively with the vertices in $\{w_r, \ldots, w_0\}$. After the swaps are complete the new index of x is $I'_e(x)$ and all vertices in F_e are higher than x, so we have:

$$I_e(x) - I'_e(x) = \sum_{y \in \{w_r, \ldots, w_0\}} d(x, y) \tag{1}$$

Repeat this process for the next vertex (v_{q-1}) and so on until all vertices in R_e are less than all vertices in F_e. Since the swaps only use existing indices of $F_e \cup R_e$ in I^{-1} and since the relative order of the vertices in each set is preserved, it follows that the final order of the vertices is: $v_q, v_{q-1}, \ldots, v_0, w_0, \ldots, w_{p-1}, w_p$. We can bound the total movement of the vertices as follows. Let π_e be the permutation due to arc e; $(x, y) \in \pi_e$ means that the pair (x, y) is swapped in π_e.

$$\sum_{x \in V} |I_e(x) - I'_e(x)| = \sum_{x \in R_e} (I_e(x) - I'_e(x)) + \sum_{y \in F_e} (I'_e(y) - I_e(y)) \tag{2}$$

$$= 2 \sum_{x \in R_e} (I_e(x) - I'_e(x)) \tag{3}$$

$$= 2 \sum_{x \in R_e} \sum_{y:(x,y) \in \pi_e} d(x, y) \tag{4}$$

$$= 2 \sum_{(x,y) \in \pi_e} d(x, y).$$

Equation 3 follows from 2 because $\sum_{x \in V} I_e(x) = \sum_{x \in V} I'_e(x)$. Equation 4 follows from Equation 1. Since each pair (x, y) is swapped at most once over all permutations, the above sum is identical to the result of Ajwani et al. within a factor of 2 and we have $\sum_{x \in V} |I_e(x) - I'_e(x)| = 2 \sum_{(x,y) \in \pi_e} d(x, y) = O(n^{2.5})$. □

6 Lower Bounds and Other Issues

A natural question to ask is whether the time bounds we have obtained for our algorithms are tight, and more generally whether there are faster algorithms for either sparse or dense graphs or both. For the sparse case, Katriel and Bodlaender [11] give a class of examples on which our soft threshold algorithm takes $\Omega(m^{1/2})$ time per arc addition; thus our analysis is tight. For the dense case, our topological search algorithm takes $\Omega(n^2/m)$ time per arc addition in the worst case by a general lower bound below. We do not know whether our bound of $O(n^{2.5}/m)$ for this algorithm is tight, although the solution of the LP used in the analysis is $\Theta(n^{2.5})$: there may be additional constraints in the behavior of the algorithm that are not captured by the LP.

More generally, Ramalingam and Reps [23] gave a class of examples in which $n-1$ arc additions force $\Omega(n \log n)$ vertex reorderings, no matter what topological order is maintained. Katriel [10] gave a class of examples on which any algorithm that (1) only reorders vertices within the affected region and (2) maintains the vertex order as an explicit mapping from the vertices to $\{1, \ldots, n\}$ must do $\Omega(n^2)$ vertex reorderings for n arc additions. Our topological search algorithm is subject to this bound, although our soft threshold algorithm is not. We have obtained the following related result:

Theorem 6. *There is a class of examples on which any algorithm that only reorders vertices within the affected region must do $\Omega(nm^{1/2})$ vertex reorderings for n arc additions.*

Proof. Omitted. See [7]. □

All existing algorithms are subject to this bound. The theorem implies that our soft threshold algorithm is within a constant factor of minimum-time on sparse graphs ($m = O(n)$) among algorithms that reorder only within the affected region.

If both additions and deletions are allowed, there is no known solution for either the topological ordering or cycle detection problem better than running an $O(m)$-time static algorithm after each graph change. There has been quite a bit of work on the harder problem of maintaining full reachability information for a dynamic graph. See [24, 25].

We have used amortized running time as our measure of efficiency. An alternative way to measure efficiency is to use an incremental competitive model [22], in which the time spent to handle an arc addition is compared against the minimum work that must be done by any algorithm, given the same current topological order and the same arc addition. The minimum work that must be done is the minimum number of vertices that must be reordered, which is the measure that Ramalingam and Reps used in their lower bound. But no existing algorithm handles an arc addition in time polynomial in the minimum number of vertices that must be reordered. To obtain positive results, some researchers have measured the performance of their algorithms against the minimum sum of degrees of vertices that must be reordered [2] or a more-refined measure that counts out-degrees of forward vertices and in-degrees of backward vertices [20]. For these models, appropriately balanced forms of ordered search are competitive to within a logarithmic factor [2, 20]. In such a model, our sparse-efficient algorithm is competitive to within a constant factor.

Alpern et al. and Pearce and Kelly consider batched arc additions as well as single arc additions. Generalizing compatible search and topological search to efficiently handle batched arc additions is a topic for future work.

Acknowledgement

The last author thanks Deepak Ajwani for his presentation at the 2007 Data Structures Workshop at Bertinoro that motivated the work in Sections 2 and 3.

References

1. D. Ajwani, T. Friedrich, and U. Meyer. An $O(n^{2.75})$ algorithm for online topological ordering. In *SWAT 2006*, volume 4059, pages 53–64, 2006.
2. B. Alpern, R. Hoover, B. K. Rosen, P. F. Sweeney, and F. K. Zadeck. Incremental evaluation of computational circuits. In *SODA 1990*, pages 32–42, 1990.
3. F. Belik. An efficient deadlock avoidance technique. *IEEE Trans. on Comput.*, 39(7), 1990.
4. M. A. Bender, R. Cole, E. D. Demaine, M. Farach-Colton, and J. Zito. Two simplified algorithms for maintaining order in a list. In *ESA 2002*, volume 2461, pages 152–164, 2002.
5. M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *J. of Comput. and Syst. Sci.*, 7(4):448–461, 1973.
6. P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *STOC 1987*, pages 365–372, 1987.

7. B. Haeupler, S. Sen, and R. E. Tarjan. Incremental topological ordering and strong component maintenance, 2008.
8. Y. Han and M. Thorup. Integer sorting in $O(n\sqrt{\log\log n})$ expected time and linear space. In *FOCS 2002*, pages 135–144, 2002.
9. F. Harary, R. Z. Norman, and D. Cartwright. *Structural Models : An Introduction to the Theory of Directed Graphs*. John Wiley & Sons, 1965.
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. *ACM Trans. on Algor.*, 2(3):364–379, 2006.
12. T. Kavitha and R. Mathew. Faster algorithms for online topological ordering, 2007.
13. D. E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, 1973.
14. D. E. Knuth and J. L. Szwarcfiter. A structured program to generate all topological sorting arrangements. *Inf. Proc. Lett.*, 2(6):153–157, 1974.
15. H.-F. Liu and K.-M. Chao. A tight analysis of the Katriel-Bodlaender algorithm for online topological ordering. *Theor. Comput. Sci.*, 389(1-2):182–189, 2007.
16. H.-F. Liu and K.-M. Chao. An $\tilde{O}(n^{2.5})$-time algorithm for online topological ordering, 2008.
17. A. Marchetti-Spaccamela, U. Nanni, and H. Rohnert. On-line graph algorithms for incremental compilation. In *WG 1993*, volume 790, pages 70–86. 1993.
18. A. Marchetti-Spaccamela, U. Nanni, and H. Rohnert. Maintaining a topological order under edge insertions. *Inf. Proc. Lett.*, 59(1):53–58, 1996.
19. S. M. Omohundro, C.-C. Lim, and J. Bilmes. The Sather language compiler/debugger implementation. Technical Report TR-92-017, International Computer Science Institute, Berkeley, 1992.
20. D. J. Pearce and P. H. J. Kelly. A dynamic topological sort algorithm for directed acyclic graphs. *J. of Exp. Algorithmics*, 11:1.7, 2006.
21. D. J. Pearce, P. H. J. Kelly, and C. Hankin. Online cycle detection and difference propagation for pointer analysis. In *SCAM 2003*, pages 3–12, 2003.
22. G. Ramalingam and T. W. Reps. On the computational complexity of incremental algorithms. Technical Report CS-TR-1991-1033, University of Wisconsin-Madison, 1991.
23. G. Ramalingam and T. W. Reps. On competitive on-line algorithms for the dynamic priority-ordering problem. *Inf. Proc. Lett.*, 51(3):155–161, 1994.
24. L. Roditty and U. Zwick. Improved dynamic reachability algorithms for directed graphs. In *FOCS 2002*, pages 679–688, 2002.
25. L. Roditty and U. Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In *STOC 2004*, pages 184–191, 2004.
26. A. Schönhage, M. Paterson, and N. Pippenger. Finding the median. *J. of Comput. and Syst. Sci.*, 13(2):184–199, 1976.
27. O. Shmueli. Dynamic cycle detection. *Information Processing Letters*, 17(4):185–188, 1983.
28. R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. on Comput.*, 1(2):146–160, 1972.
29. M. Thorup. Integer priority queues with decrease key in constant time and the single source shortest paths problem. *J. of Comput. Syst. Sci.*, 69(3):330–353, 2004.
30. P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Inf. Proc. Lett.*, 6(3):80–82, 1977.
31. P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.