

An $O(n^{2.75})$ algorithm for incremental topological ordering

DEEPAK AJWANI

TOBIAS FRIEDRICH

Max-Planck-Institut für Informatik, Germany

and

ULRICH MEYER

J.W. Goethe University Frankfurt/Main, Germany

We present a simple algorithm which maintains the topological order of a directed acyclic graph with n nodes under an online edge insertion sequence in $O(n^{2.75})$ time, independent of the number m of edges inserted. For dense DAGs, this is an improvement over the previous best result of $O(\min\{m^{\frac{3}{2}} \log n, m^{\frac{3}{2}} + n^2 \log n\})$ by Katriel and Bodlaender. We also provide an empirical comparison of our algorithm with other algorithms for incremental topological sorting.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Computations on discrete structures*; E.1 [**Data Structures**]: Graphs and networks; G.2.2 [**Discrete Mathematics**]: Graph Theory—*Graph algorithms*

General Terms: Algorithms, Design, Theory

Additional Key Words and Phrases: Dynamic algorithms, graphs, online algorithms, topological order

1. INTRODUCTION

A topological order T of a given directed acyclic graph (DAG) $G = (V, E)$ (with $n := |V|$ and $m := |E|$) is a linear ordering of its nodes such that for all directed

Research partially supported by DFG grant ME 2088/1-3, and by the center of massive data algorithmics (MADALGO) funded by the Danish National Research Foundation.

A preliminary version of this article appeared in *Proceedings of the 10th Scandinavian Workshop on Algorithm Theory (SWAT 2006)*, volume 4059 of LNCS, pages 53–64, Springer, 2006.

Authors' addresses: D. Ajwani and T. Friedrich, Max-Planck-Institut für Informatik, Campus E1 4, 66123 Saarbrücken, Germany, e-mail: firstname.lastname@mpi-inf.mpg.de; U. Meyer, Institute for Computer Science, J.W. Goethe University, 60325 Frankfurt/Main, Germany, e-mail: umeyer@cs.uni-frankfurt.de

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 2008 ACM /2008/-0001 \$5.00

paths from $x \in V$ to $y \in V$ ($x \neq y$), it holds that $T(x) < T(y)$. There exist well known algorithms for computing the topological ordering of a DAG in $O(m+n)$ in an offline setting [Tarjan 1972; Knuth and Szwarcfiter 1974].

In the online variant of this problem, the edges of the DAG are not known in advance but are given one at a time. Each time an edge is added to the DAG, we are required to update the bijective mapping T .

Incremental topological ordering is required for incremental evaluation of computational circuits [Alpern et al. 1990] and incremental compilation [Marchetti-Spaccamela et al. 1993; Omohundro et al. 1992] where a dependency graph between modules is maintained to reduce the amount of recompilation performed when an update occurs. It is also used as an online cycle detection routine in pointer analysis [Pearce et al. 2003]. In this problem, one wants to discover the first edge which introduces a cycle in an arbitrary sequence of edges. Till now, the best known algorithm for online cycle detection is to compute the incremental topological ordering.

The naïve way of computing an incremental topological order each time from scratch with the offline algorithm takes $O(m^2 + mn)$ time. Marchetti-Spaccamela, Nanni, and Rohnert [1996] (MNR) gave an algorithm that can insert m edges in $O(mn)$ time. Alpern, Hoover, Rosen, Sweeney, and Zadeck [1990] (AHSZ) proposed an algorithm which runs in $O(\|\delta\| \log \|\delta\|)$ time per edge insertion with $\|\delta\|$ measuring the minimum sum of all nodes which have to be updated and of all edges incident to these nodes. Note that not all edges of this subgraph need to be visited and hence even $O(\|\delta\|)$ time per insertion is not optimal. However, there is no analysis of AHSZ for a sequence of edge insertions. Katriel and Bodlaender [2006] (KB) analyzed a variant of the AHSZ algorithm and obtained an upper bound of $O(\min\{m^{\frac{3}{2}} \log n, m^{\frac{3}{2}} + n^2 \log n\})$ for a general edge sequence. In addition, they show that their algorithm runs in time $O(m \cdot k \cdot \log^2 n)$ for a DAG for which the underlying undirected graph has a treewidth k . Also, they give an $O(n \log n)$ algorithm for DAGs whose underlying undirected graph is a tree. Liu and Chao [2007] give a tight analysis of the algorithm KB by showing that it runs in time $\Theta(m^{3/2} + mn^{1/2} \log n)$. The algorithm by Pearce and Kelly [2006] (PK) empirically outperforms the other algorithms for random edge insertions leading to sparse random DAGs, although its worst-case runtime is inferior to KB. Recently, Ajwani and Friedrich [2007] have also proven an expected runtime of $O(n^2 \text{polylog}(n))$ under insertion of the edges of a complete DAG in a random order for AHSZ, KB, and PK. The only non-trivial lower bound for this problem is by Ramalingam and Reps [1994], who show that an adversary can force any algorithm maintaining explicit labels to need $\Omega(n \log n)$ time complexity for inserting $n - 1$ edges.

We propose a simple algorithm that works in $O(n^{2.75} \sqrt{\log n})$ time and $O(n^2)$ space, thereby improving upon the results of Katriel and Bodlaender for dense DAGs. With some simple modifications in our data structure, we can get $O(n^{2.75})$ time with $O(n^{2.25})$ space or $O(n^{2.75})$ *expected* time with $O(n^2)$ space. We also demonstrate empirically that this algorithm clearly outperforms KB, MNR, AHSZ, and PK on a certain class of hard sequences of edge insertions, while being at most a factor of 2-4 away on random edge sequences leading to complete DAGs.

Our algorithm is dynamic, as it also supports deletion. However, our analysis holds only for a sequence of insertions. Note that our algorithm can also be used for online cycle detection in graphs. Moreover, it permits an arbitrary starting point, which makes a hybrid approach possible, i. e., using the PK or KB algorithm for sparse graphs and ours when the graphs become dense.

The rest of this paper is organized as follows. In Section 2, we describe the algorithm and the data structures involved. In Section 3, we give the correctness argument for our algorithm, followed by an analysis of its runtime in Sections 4 and 5. The details of our implementation and an empirical comparison with other algorithms follow in Section 6.

2. ALGORITHM

We keep the current topological order as a bijective function $T: V \rightarrow [1..n]$. If we start with an empty graph, we can initialize T with an arbitrary permutation, otherwise T is the topological order of the initial graph, computed offline. In this and the subsequent sections, we will use the following notations: $d(u, v)$ denotes $|T(u) - T(v)|$, $u < v$ is a short form of $T(u) < T(v)$, $u \rightarrow v$ denotes an edge from u to v , and $u \rightsquigarrow v$ expresses that v is reachable from u . Note that $u \rightsquigarrow u$, but *not* $u \rightarrow u$.

Figure 1 gives the pseudo code of our algorithm. Throughout the process of inserting new edges, we maintain some data structures which are dependent on the current topological order. Inserting a new edge (u, v) is done by calling `INSERT(u, v)`. If $v > u$, we do not change anything in the current topological order and simply insert the edge into the graph data structure. Otherwise, we call `REORDER` to update the topological order as well as the data structures dependent on it. As we will prove in Theorem 4, detecting $v = u$ in a call of `REORDER(u, v)` indicates a cycle. If $v < u$, we first collect the sorted sets A and B . A is the set of out-neighbors of v whose topological order is not greater than $T(u)$. Analogously, B is the set of in-neighbors of u whose topological order is not less than $T(v)$. If both A and B are empty, we swap the topological order of the two nodes and update the data structures. Otherwise, we recursively call `REORDER` until everything inside is topologically ordered. To make these recursive calls efficient, we first merge the sorted sets $\{v\} \cup A$ and $B \cup \{u\}$ and (using this merged list) compute the set $\{u' : (u' \in B \cup \{u\}) \wedge (u' \geq v')\}$ for each node $v' \in \{v\} \cup A$. The collection of sets A and B and the update operations are described in more detail after the data structures have been introduced.

Data structure

We store the current topological order as a set of two arrays by maintaining the bijective mapping T and its inverse T^{-1} . This ensures that finding $T(u)$ and $T^{-1}(i)$ are constant time operations.

The graph itself is stored as an array of vertices. For each vertex we maintain

```

INSERT( $u, v$ )
  ▷ Insert edge  $(u, v)$  and calculate new topological order
  1 if  $v \leq u$  then REORDER( $u, v$ )
  2 insert edge  $(u, v)$  in graph

REORDER( $u, v$ )
  ▷ Reorder nodes between  $u$  and  $v$  if  $v \leq u$ 
  1 if  $u = v$  then report detected cycle and quit
  2  $A := \{w : v \rightarrow w \text{ and } w \leq u\}$ 
  3  $B := \{w : w \rightarrow u \text{ and } v \leq w\}$ 
  4 if  $A = \emptyset$  and  $B = \emptyset$ 
    then ▷ Correct the topological order
  5       swap  $T(u)$  and  $T(v)$ 
  6       update the data structure
  7 else ▷ Reorder node pairs between  $v$  and  $u$ 
  8   for  $v' \in \{v\} \cup A$  in decreasing topological order
  9     for  $u' \in B \cup \{u\} \wedge v' \leq u'$  in increasing topological order
      REORDER( $u', v'$ )

```

Fig. 1. Our algorithm

two adjacency lists, which keep the incoming and outgoing edges separately. Each adjacency list is stored as an array of buckets of vertices. Each bucket contains at most t nodes for a fixed t . Depending on the concrete implementation of the buckets, the parameter t is later chosen to be approximately $n^{0.75}$ so as to balance the number of inserts and deletes from the buckets and the extra edges touched by the algorithm. The i -th bucket ($i \geq 0$) of a node x contains all adjacent nodes y with $i \cdot t < d(x, y) \leq (i + 1) \cdot t$. The nodes of a bucket are stored with node index (and not topological order) as their key. This has the advantage that there is no change necessary if two nodes that lie in the same bucket are swapped. The bucket can be kept as a balanced binary tree, as an array of n -bits, or as a hash-table of a universal hashing function. The only requirement for the bucket data structure is that it should provide efficient support for the following three operations:

- (1) *Insert*: Insert an element in a given bucket.
- (2) *Delete*: Given an element and a bucket, find out if that element exists in that bucket. If yes, delete the element from there and return 1. Else, return 0.
- (3) *Collect-all*: Copy all the elements from the bucket to some vector.

Depending on how we choose to implement the buckets, we get different runtimes. This will be discussed in Section 5. We will now discuss how we do the insertion of an edge, computation of A and B , and updating the data structure under swapping of nodes in terms of the above three basic operations.

Inserting an edge (u, v) means inserting node v in the forward adjacency list of u and u in the backward adjacency list of v . This requires $O(1)$ bucket inserts.

For given u and v , the set $A := \{w: v \rightarrow w \text{ and } w < u\}$ sorted according to the current topological order can be computed from the adjacency list of v by sorting all nodes of the first $\lceil d(u, v)/t \rceil$ outgoing buckets and choosing all w with $w < u$. This can be done by $O(d(u, v)/t)$ collect-all operations on buckets. This means traversing all elements of A as well as all elements of the $\lceil d(u, v)/t \rceil$ -th outgoing bucket. Overall $O(|A| + t)$ elements are visited. These elements are integers in the range $\{1 \dots n\}$ and can be sorted in $O(|A| + t)$ time using a two-pass radix sort algorithm since t is chosen such that $t \geq n^{0.75}$. The set B is computed likewise from the incoming edges.

When we swap two nodes u and v , we need to update the adjacency lists of u and v as well as that of all nodes w that are adjacent to u and/or v . First, we show how to update the adjacency lists of u and v . If $d(u, v) > t$, we build their adjacency lists from scratch. Otherwise, the new bucket boundaries will differ from the old boundaries by $d(u, v)$ and at most $d(u, v)$ nodes will need to be transferred between any pair of consecutive buckets. The total number of transfers are therefore bounded by $d(u, v)\lceil n/t \rceil$. Determining whether a node should be transferred can be done in $O(1)$ using the inverse mapping T^{-1} and as noted above, a transfer can be done in $O(1)$ bucket inserts and deletes. Hence, updating the adjacency lists of u and v needs at most $\min\{n, d(u, v)\lceil n/t \rceil\}$ bucket inserts and deletes.

Let w be a node which is adjacent to u or v . Its adjacency list needs to be updated only if u and v are in different buckets. This corresponds to w being in different buckets of the adjacency lists of u and v . Therefore, the number of nodes to be transferred between different buckets for maintaining the adjacency lists of all w 's is the same as the number of nodes that need to be transferred for maintaining the adjacency lists of u and v , i. e., $\min\{n, d(u, v)\lceil n/t \rceil\}$.

Updating the mappings T and T^{-1} after such a swap is trivial and can be done in constant time. Thus, we conclude that swapping nodes u and v can be done by $O(d(u, v)\lceil n/t \rceil)$ bucket inserts and deletes.

3. CORRECTNESS

In this section we will show the following theorem.

THEOREM 1. *The above algorithm returns a valid topological order after each edge insertion.*

PROOF. For a graph with no edges, any ordering is a correct topological order, and therefore, the theorem is trivially correct. Assuming that we have a valid topological order of a graph G , we show that when inserting a new edge (u, v) using $\text{INSERT}(u, v)$, our algorithm maintains the correct topological order of $G' := G \cup \{(u, v)\}$. If $u < v$, this is trivial.

We need to prove that $x < y$ for all nodes x, y of G' with $x \rightsquigarrow y$. If there was a path $x \rightsquigarrow y$ in G , Lemma 2 gives $x < y$. Otherwise (if there is no $x \rightsquigarrow y$ in G), the

path $x \rightsquigarrow y$ must have been introduced to G' by the new edge (u, v) . Hence $x < y$ in G' by Lemma 3 since there is $x \rightsquigarrow u \rightarrow v \rightsquigarrow y$ in G' . \square

LEMMA 2. *Given a DAG G and a valid topological order, if $u \rightsquigarrow v$ and $u < v$, then all subsequent calls to REORDER will maintain $u < v$.*

PROOF. Let us assume the contrary. Consider the first call of REORDER which for a node pair u, v with $u \rightsquigarrow v$ and $u < v$ leads to $u > v$. Either this call led to swapping u and v with $v \leq w$ or it caused swapping w and v with $w \leq u$. Note that in our algorithm, a call of REORDER(u, v) leads to a swapping only if $A = \emptyset$ and $B = \emptyset$. Assuming that it was the first case (swapping u and w) caused by the call to REORDER(u, w), $A = \emptyset$. However, since u, v is the first such pair to get violated, $x \in A$ for an x with $u \rightarrow x \rightsquigarrow v$, leading to a contradiction. The other case is proved analogously. \square

LEMMA 3. *Given a DAG G with $v \rightsquigarrow y$ and $x \rightsquigarrow u$, a call of REORDER(u, v) will ensure that $x < y$.*

PROOF. Consider the recursion tree of a call to REORDER, in which the recursive calls emanating in lines 7 and 8 are its children. The proof follows by induction on the recursion tree height of REORDER(u, v). For leaf nodes (calls of REORDER with zero recursion tree height) of the recursion tree, $A = B = \emptyset$. If $x < y$ before this call, Lemma 2 ensures that $x < y$ will still hold. Otherwise, $y := v$ and $x := u$. The swapping of u and v in line 5 gives $x < y$.

We assume this lemma to be true for calls of REORDER up to a certain recursion tree height and consider a call with a higher recursion tree. If $A \neq \emptyset$, then there is a \tilde{v} such that $v \rightarrow \tilde{v} \rightsquigarrow y$, otherwise $\tilde{v} := v = y$. If $B \neq \emptyset$, then there is a \tilde{u} such that $x \rightsquigarrow \tilde{u} \rightarrow u$, otherwise $\tilde{u} := u = x$. Hence $\tilde{v} \rightsquigarrow y < x \rightsquigarrow \tilde{u}$. The **for**-loops of lines 7 and 8 will call REORDER(\tilde{u}, \tilde{v}). By the inductive hypothesis, this will ensure $x < y$. According to Lemma 2, further calls to REORDER will maintain $x < y$. \square

THEOREM 4. *The algorithm detects a cycle if and only if there is a cycle in the given edge sequence.*

PROOF. “ \Rightarrow ”: First, we show that within a call to INSERT(u, v), there are paths $v \rightsquigarrow v'$ and $u' \rightsquigarrow u$ for each recursive call to REORDER(u', v'). This is trivial for the first call to REORDER and follows immediately by the definition of A and B for all subsequent recursive calls to REORDER. This implies that if the algorithm indicates a cycle in line 1 of REORDER, there is indeed a cycle $u \rightarrow v \rightsquigarrow v' = u' \rightsquigarrow u$. In fact, the cycle itself can be computed using the recursion stack of the current call to REORDER.

“ \Leftarrow ”: Consider the edge (u, v) of the cycle $v \rightsquigarrow u \rightarrow v$ inserted last. Since $v \rightsquigarrow u$ before the insertion of this edge, the topological order computed will satisfy $v < u$ (Theorem 1) and therefore, REORDER(u, v) would be called. In fact, all edges in

the path $v \rightsquigarrow u$ will obey the current topological ordering and by Lemma 2, it will remain so for all subsequent calls of REORDER. We prove by induction on the number of nodes in the path $v \rightsquigarrow u$ (including u and v) that whenever $v \rightsquigarrow u$ and REORDER(u, v) is called, it detects the cycle. A call of REORDER(u', v') with $u' = v'$ or REORDER(u', v') with $v' \rightarrow u'$ clearly reports a cycle. Consider a path $v \rightarrow x \rightsquigarrow y \rightarrow u$ of length $k > 2$ and the call of REORDER(u, v). As noted before, $v < x \leq y < u$ before the call to REORDER(u, v). Hence $x \in A$ and $y \in B$ and a call to REORDER(y, x) will be made in the for loop of lines 7 and 8. As $y \rightsquigarrow x$ has $k - 2$ nodes in the path, the call to REORDER(y, x) (by our inductive hypothesis) will detect the cycle. \square

4. RUNTIME

The following theorem is the main result of this section.

THEOREM 5. *Incremental topological ordering can be maintained while processing any sequence of edge insertions using $O(n^{3.5}/t)$ bucket inserts and deletes, $O(n^3/t)$ bucket collect-all operations collecting $O(n^2t)$ elements, and $O(n^{2.5} + n^2t)$ operations.*

PROOF. Consider the pseudo code in Figure 1. Since there can be a maximum of $n(n - 1)/2$ edges inserted in a DAG, there are $O(n^2)$ calls of INSERT. Inserting an edge in the graph involves $O(1)$ bucket operations and therefore, the total cost of Line 2 of INSERT is $O(n^2)$.

Lemma 8 shows that REORDER is called $O(n^2)$ times. Line 1 of REORDER requires $O(1)$ operations per call of REORDER, except the one time it does encounter a cycle (when it requires $O(n)$ time). Lemma 10 shows that the calculation of the sets A and B over all calls of REORDER can be done by $O(n^3/t)$ bucket collect-all operations touching $O(n^2t)$ edges, and $O(n^{2.5} + n^2t)$ operations. Lines 4 and 5 require $O(1)$ operations per call of REORDER. In Lemma 12, we prove that all the updates can be done by $O(n^{3.5}/t)$ bucket inserts and deletes.

For lines 7 and 8 of the pseudo-code, we first merge the two sorted sets A and B . This takes $O(|A| + |B|)$ operations. For a particular node $v' \in \{v\} \cup A$, we can compute the set $V' = \{u' : (u' \in B \cup \{u\}) \wedge (u' \geq v')\}$ (as required by line 8) using this merged set in complexity $O(1 + |V'|)$, which is also the number of calls of REORDER emanating for this particular node. Summing over the entire for loop of line 7, the total complexity of lines 7 and 8 is $O(|A| + |B| + \text{number of calls of REORDER emanating from here})$. Since by Lemma 9, the summation of $|A| + |B|$ over all calls of REORDER is $O(n^2)$ and by Lemma 8, the total number of calls to REORDER is also $O(n^2)$, we get a total of $O(n^2)$ operations for lines 7 and 8. The theorem follows by simply adding the complexity of each line. \square

LEMMA 6. *REORDER is local, i. e., a call to REORDER(u, v) does not affect the topological ordering of nodes w such that either $w < v$ or $w > u$ just before the call was made.*

PROOF. This lemma can be proven by induction on the level of the recursion tree of a call to $\text{REORDER}(u, v)$. For the leaf node of the recursion tree, $|A| = |B| = 0$ and the topological order of u and v is swapped, not affecting the topological ordering of any other node.

We assume this lemma to be true up to a certain tree level. To see that it is also valid for one level higher, note that the arrays A and B contain elements w such that $v < w < u$. Since each call of REORDER in the **for**-loop of line 7 and 8 is from an element of A to an element of B and all of these calls are themselves local by our induction hypothesis, this call of REORDER is also local. \square

LEMMA 7. *If two nodes are swapped in a call of REORDER , their relative order will remain unchanged in the future.*

PROOF. Let us assume, two nodes u' and v' are swapped within one of the recursive calls of REORDER invoked by $\text{INSERT}(u, v)$. After the insertion of edge (u, v) , there is a path $u' \rightsquigarrow u \rightarrow v \rightsquigarrow v'$. Therefore, by Lemma 2 the relative order of u' and v' will not be changed in any subsequent call of INSERT .

It remains to prove that also within the recursion tree of $\text{REORDER}(u, v)$, the relative order of u' and v' will not be changed after they have been swapped. This is ensured by the order in which the two **for**-loops in lines 7 and 8 iterate since there can be no calls to $\text{REORDER}(u', w)$ with $w > v'$ or $\text{REORDER}(w, v')$ with $u < u'$ after the call of $\text{REORDER}(u', v')$. \square

LEMMA 8. *REORDER is called $O(n^2)$ times.*

PROOF. As we have proven that the algorithm is correct in Section 3, we now know that for each pair (u, v) the following holds: If $\text{REORDER}(u, v)$ is called, then $v \leq u$ holds before and $u \leq v$ holds afterwards. As by Lemma 7 this implies that $\text{REORDER}(u, v)$ can only be called once for each pair (u, v) , the number of calls to REORDER can be upper bounded by n^2 . \square

LEMMA 9. *The summation of $|A| + |B|$ over all calls of REORDER is $O(n^2)$.*

PROOF. Consider arbitrary nodes u and v' . We prove that for all $v \in V$, $v' \in A$ happens only once over all calls of $\text{REORDER}(u, v)$. This proves that $\sum |A| \leq n$, for all such calls of $\text{REORDER}(u, v)$. Therefore, summing up for all $u \in V$, $\sum |A| \leq n^2$ over all calls of REORDER .

In order to see that for all $v \in V$, $v' \in A$ happens only once over all calls of $\text{REORDER}(u, v)$, consider the first such call. Since $v' \in A$, $v' < u$ and $v \rightarrow v'$ before the call was made. By Lemma 3, $u < v'$ after this call and hence, $v' \notin A$ for any call of REORDER afterwards. As for calls within the recursive substructure of the first call, the order in which these calls are made ensures that there will be no calls of $\text{REORDER}(u, w)$ for any $w < v'$ before $\text{REORDER}(u, v')$ and since $u < v'$ after $\text{REORDER}(u, v')$, $v' \notin A$ for $\text{REORDER}(u, w)$.

Analogously, it can be proven that for arbitrary nodes v and v' and for all $u \in V$, $v' \in B$ happens only once over all calls of $\text{REORDER}(u, v)$. The proof for $\sum |B| \leq n^2$ follows similarly and it completes the proof of this lemma. \square

LEMMA 10. *Calculating the sorted sets A and B over all calls of REORDER can be done by $O(n^3/t)$ bucket collect-all operations touching a total of $O(n^2t)$ elements and $O(n^{2.5} + n^2t)$ operations for sorting these elements.*

PROOF. Consider the calculation of set A in a call of $\text{REORDER}(u, v)$. As discussed before in Section 2, we look at the out adjacency list of u , stored in the form of buckets. In particular, we will need $O(d(u, v)/t)$ bucket collect-all operations touching $O(|A| + t)$ elements to calculate A . The additional worst-case factor of t stems from the last bucket visited. Summing up over all calls of REORDER , we get $O(\sum d(u, v)/t)$ collect-all's touching $\sum(|A| + |B| + t)$ elements. Since $d(u, v) \leq n$ for every call of $\text{REORDER}(u, v)$ and there are $O(n^2)$ calls of REORDER (Lemma 8), there are $O(n^3/t)$ bucket collect-all operations. Also, since $\sum(|A| + |B|) = O(n^2)$ by Lemma 9, the total number of elements touched is $O(n^2 + \sum t) = O(n^2t)$. Since the keys are in the range $\{1 \dots n\}$, we can use a two-pass radix sort to sort the elements collected from the buckets. The total sorting time over all calls of REORDER is $\sum(2(|A| + t) + \sqrt{n}) + \sum(2(|B| + t) + \sqrt{n}) = O(n^{2.5} + n^2t)$. \square

LEMMA 11. $\sum d(u, v) = O(n^{5/2})$ where the summation is taken over all calls of $\text{REORDER}(u, v)$ in which u and v are swapped.

PROOF. Let T^* denote the final topological ordering and

$$X(T^*(u), T^*(v)) := \begin{cases} d(u, v) & \text{if } \text{REORDER}(u, v) \text{ leads to a swapping} \\ 0 & \text{otherwise} \end{cases}$$

As Lemma 7 implies that each node pair is swapped at most once, the variable $X(i, j)$ is clearly defined. Next, we model a few linear constraints on $X(i, j)$, formulate it as a linear program and use this LP to prove that $\max\{\sum_{i,j} X(i, j)\} = O(n^{5/2})$. By definition of $d(u, v)$ and $X(i, j)$,

$$0 \leq X(i, j) \leq n \text{ for all } i, j \in [1 \dots n].$$

For $j \leq i$, the corresponding edges $(T^{*-1}(i), T^{*-1}(j))$ go backwards and thus are never inserted at all. Consequently,

$$X(i, j) = 0 \text{ for all } j \leq i.$$

Now consider an arbitrary node u , which is finally at position i , i.e., $T^*(u) = i$. Over the insertion of all edges, this node has been moved left and right via swapping with several other nodes. Strictly speaking, it has been swapped right with nodes at final positions $j > i$ and has been swapped left with nodes at final positions $j < i$. Hence, the overall movement to the right is $\sum_{j>i} X(i, j)$ and to left is $\sum_{j<i} X(j, i)$. Since the net movement (difference between the final and the initial position) must

be less than n ,

$$\sum_{j>i} X(i, j) - \sum_{j<i} X(j, i) \leq n \text{ for all } 1 \leq i \leq n.$$

Putting all the constraints together, we aim to solve the following linear program.

$$\max \sum_{\substack{1 \leq i \leq n \\ 1 \leq j \leq n}} X(i, j) \text{ such that}$$

- (i) $X(i, j) = 0$ for all $1 \leq i \leq n$ and $1 \leq j \leq i$,
- (ii) $0 \leq X(i, j) \leq n$ for all $1 \leq i \leq n$ and $i < j \leq n$,
- (iii) $\sum_{j>i} X(i, j) - \sum_{j<i} X(j, i) \leq n$ for all $1 \leq i \leq n$.

Note that these are necessary constraints, but not sufficient. But this is enough for our purpose as an upper bound to the solution of this LP will give an upper bound for the $\sum X(i, j)$ in our algorithm. In order to prove the upper bound on the solutions of this LP, we consider the dual problem

$$\min \left[n \sum_{\substack{0 \leq i < n \\ i < j < n}} Y(i \cdot n + j) + n \sum_{0 \leq i < n} Y(n^2 + i) \right] \text{ such that}$$

- (i) $Y(i \cdot n + j) \geq 1$ for all $0 \leq i < n$ and $0 \leq j \leq i$,
- (ii) $Y(i \cdot n + j) + Y(n^2 + i) - Y(n^2 + j) \geq 1$ for all $0 \leq i < n$ and $j > i$,
- (iii) $Y(i) \geq 0$ for all $0 \leq i < n^2 + n$.

and the following feasible solution for the dual:

$$\begin{aligned} Y(i \cdot n + j) &= 1 && \text{for all } 0 \leq i < n \text{ and } 0 \leq j \leq i, \\ Y(i \cdot n + j) &= 1 && \text{for all } 0 \leq i < n \text{ and } i < j \leq i + 1 + 2\sqrt{n}, \\ Y(i \cdot n + j) &= 0 && \text{for all } 0 \leq i < n \text{ and } j > i + 1 + 2\sqrt{n}, \\ Y(n^2 + i) &= \sqrt{n - i} && \text{for all } 0 \leq i < n. \end{aligned}$$

This solution has a value of $n^2 + 2n^{5/2} + n \sum_{i=1}^n \sqrt{i} = O(n^{5/2})$, which by the primal-dual theorem is a bound on the solution of the original LP.

In fact, it can be shown that there is a solution to primal LP whose value is $O(n^{5/2})$, namely

$$\begin{aligned} X(i, j) &= 0 && \text{for all } 0 \leq i < n \text{ and } 0 \leq j \leq i, \\ X(i, j) &= n && \text{for all } 0 \leq i < n \text{ and } i < j \leq i + \lceil \frac{\sqrt{1+8i}-1}{2} \rceil, \\ X(i, j) &= 0 && \text{for all } 0 \leq i < n \text{ and } j > i + \lceil \frac{\sqrt{1+8i}-1}{2} \rceil. \quad \square \end{aligned}$$

LEMMA 12. *Updating the data structure over all calls of REORDER requires $O(n^{3.5}/t)$ bucket inserts and deletes.*

PROOF. Our data structure requires $O(d(u, v)n/t)$ bucket inserts and deletes to swap two nodes u and v . Lemma 7 shows that each node pair is swapped at most once. Hence, summing up over all calls of $\text{REORDER}(u, v)$ where u and v are swapped, we need $O(\sum d(u, v)n/t) = O(n^{3.5}/t)$ bucket inserts and deletes using Lemma 11. \square

5. BUCKET DATA STRUCTURE

We get different runtimes and space requirements of our algorithm depending on the data structures of the buckets used:

- (a) Balanced binary trees (see e. g. Guibas and Sedgwick [1978]): Balanced binary trees give us $O(1 + \log \tau)$ time insert and delete and $O(1 + \tau)$ time collect-all operation, where τ is the number of elements in the bucket. Therefore, by Theorem 5, the total time required will be $O(n^2t + n^{3.5} \log n/t)$. Substituting $t = n^{0.75} \sqrt{\log n}$, we get a total time of $O(n^{2.75} \sqrt{\log n})$. The total space requirement will be $O(n^2)$ as a balanced binary tree needs $O(t)$ nodes for storing at most t elements.
- (b) n -bit array: A bucket that stores at most t elements can be kept as an n -bit array, where each bit is 0 or 1 depending on whether or not the element is present in the bucket. Also, we can keep a list of all elements in the bucket. To insert, we just flip the appropriate bit and insert at the end of the list. To delete, we just flip the appropriate bit. To collect all, we go through the list and for each element in the list, we check if the corresponding bit is 1 or 0. If it is 0, we also remove it from the list. This gives us constant-time insert and delete and the time for collect-all operation will be the total output size plus the total number of delete. Each delete is counted once in collect-all as we remove the corresponding element from the list after the first collect-all. By Theorem 5, the total time required will be $O(n^2t + n^{3.5}/t)$, giving us $O(n^{2.75})$ for $t = n^{0.75}$. The total space requirement will be $O(n)$ for each bucket, leading to a total of $O(n^{2.25})$ for $O(n^2/t)$ buckets.
- (c) Uniform Hashing [Östlin and Pagh 2003]: A data structure based on uniform hashing coupled with a list of elements in the bucket operated in the same way as the n -bit array will give an expected constant-time insert and delete and the same bound for collect-all as for the n -bit array. This gives an expected total time of $O(n^2t + n^{3.5}/t)$. With $t = n^{0.75}$ this yields an expected time of $O(n^{2.75})$. Since the hashing based data structure as described in [Östlin and Pagh 2003] takes only linear space, the total space requirement is $O(n^2)$.

6. EMPIRICAL COMPARISON

We conducted our experiments on a 2.4 GHz Opteron machine with 8GB of main memory running Debian GNU/Linux. For PK, MNR, and AHRSZ we used the C++/Boost based implementation of David J. Pearce (see Pearce and Kelly [2006]). For our algorithm (AFM), we implemented variant (b) of Section 5 using C++/STL. Additionally, we also implemented a local (cf. Lemma 6) variant of KB using an ordered bi-directional list data structure [Dietz and Sleator 1987]. The code of AFM and KB is available upon request. All codes were compiled using gcc 3.3 in 32-bit mode and optimization level `-O3`. The timings were measured using the `gettimeofday` function of `<sys/time.h>` and all the results are averaged over 10 runs each.

We examined all five algorithms on two classes of DAGs. First, we considered random edge insertion sequences leading to a complete DAG. This random DAG model by Barak and Erdős [1984] is similar to the well-known $G(n, m)$ random graph model of Erdős and Rényi [1959]. On a random edge sequence, all the algorithms are quite fast and none of them encounters its worst-case behavior. Therefore, we also considered a particular sequence of edges which we believe is a hard instance of the problem. This edge sequence is similar to the worst-case sequence given by Katriel and Bodlaender [2006] for their algorithm. On this sequence, KB, PK, MNR, and AHRSZ (the variant choosing the smallest permitted priority) face their worst-case of $\Omega(n^3)$ operations, while our algorithm takes $\Omega(n^{2.5})$ time complexity. This sequence of edges is depicted in Fig. 5. Let us briefly describe its structure. For a graph with n nodes, we divide the set of nodes into four blocks of different sizes: block 1 consists of nodes $[0..n/3)$, block 2 of nodes $[n/3..n/2)$, block 3 of nodes $[n/2..2n/3)$, and block 4 of nodes $[2n/3..n)$. First, we insert $n - 4$ edges such that within each block, the vertices form a directed path from left to right. Then we insert the following edges,

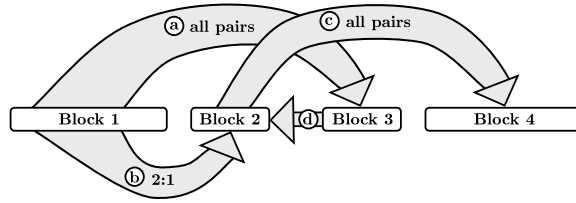


Fig. 5. Our hard-case graph

(a) $\vec{\forall} j \in [0..n/3) \quad \overleftarrow{\forall} k \in [0..n/6) : \text{add edge}(j, k + n/2)$,
 (b) $\vec{\forall} j \in [0..n/6) : \text{add edge}(2j, j + n/3) \text{ and } \text{edge}(2j + 1, j + n/3)$,
 (c) $\vec{\forall} j \in [0..n/6) \quad \overleftarrow{\forall} k \in [0..n/3) : \text{add edge}(j + n/3, k + 2n/3)$,
 (d) $\vec{\forall} j \in [0..n/6) \quad \overleftarrow{\forall} k \in [0..n/6) : \text{add edge}(j + n/2, k + n/3)$,

where $\vec{\forall}$ denotes going from left to right in the **for**-loop and $\overleftarrow{\forall}$ the other way around.

Fig. 2 shows the runtimes of the five algorithms in consideration for random edge sequences leading to complete DAGs with varying number n of vertices (and with

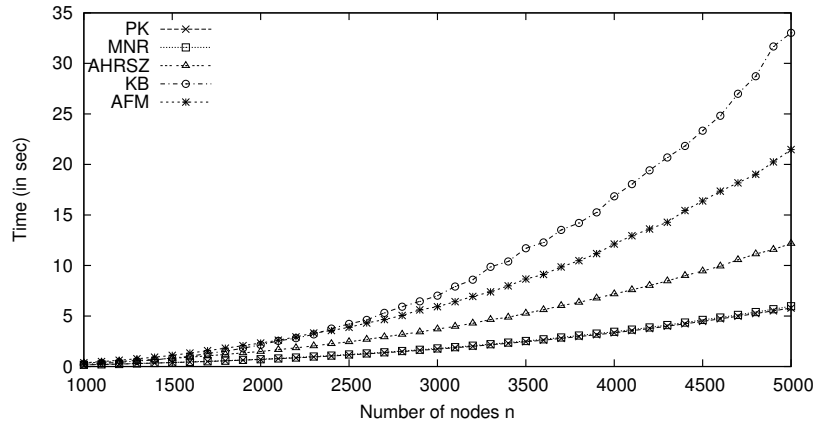


Fig. 2. Experimental data on full random graphs with varying n

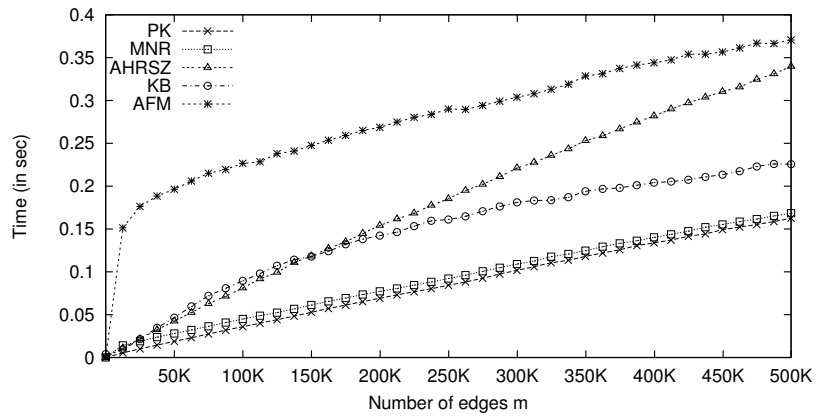


Fig. 3. Experimental data on random graphs with $n = 1000$ and varying m

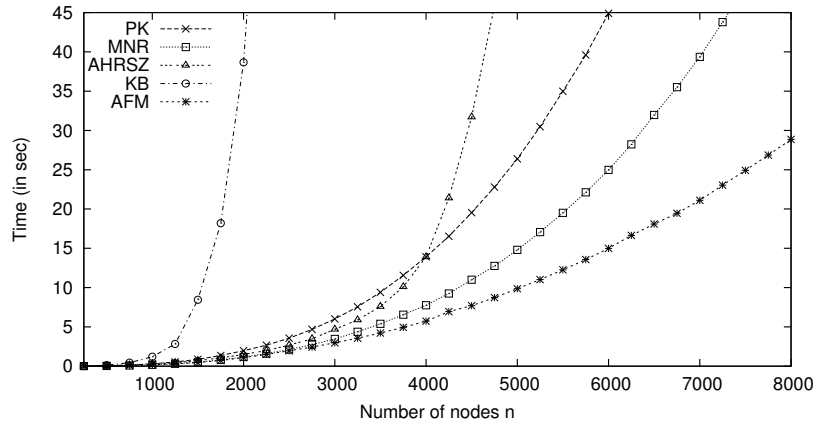


Fig. 4. Experimental data on a class of hard instances with varying n

$m = \binom{n}{2}$). We see that AFM is approximately 30% faster than KB and a constant factor of 2-4 away from AHRSZ, MNR, and PK.

Fig. 3 shows the average runtimes for random graphs with $n = 1000$ and a varying number of edges. AFM loses a lot during the insertion of the first $O(n \log n)$ edges because in this phase, updating the data structures after every swapping proves very costly. But after that, the curves between AFM and PK/MNR/KB are almost parallel, while the slope for AHRSZ is around 2 times that of AFM. For practical purposes, we believe therefore that a hybrid approach would perform best. That is, one inserts the first $O(n \log n)$ edges with either PK or KB and then inserts the remaining edges with our algorithm.

Fig. 4 shows the runtimes of the five algorithms in consideration on the class of hard edge sequences described before. The difference in asymptotic behaviour as discussed before is clear from the graph.

7. DISCUSSION

We have presented the first $o(n^3)$ algorithm for incremental topological ordering. We also implemented this new algorithm and compared it with previous approaches, showing that for certain hard examples, it outperforms PK, MNR, and AHRSZ. There is still a large gap between $\Omega(n \log n)$ lower bound for inserting $n-1$ edges, the trivial lower bound of $\Omega(m)$ for $m > n \log n$, and the upper bound of $O(\min\{m^{1.5} + n^2 \log n, m^{1.5} \log n, n^{2.75}\})$. Bridging this gap remains an open problem.

Acknowledgements

The authors are grateful to David J. Pearce for providing us his code. Also, thanks are due to the anonymous referees for valuable comments and to Seth Pettie and Saurabh Ray for helpful discussions.

REFERENCES

- AJWANI, D. AND FRIEDRICH, T. 2007. Average-case analysis of online topological ordering. In *Proceedings of the 18th International Symposium on Algorithms and Computation (ISAAC)*. Lecture Notes in Computer Science, vol. 4835. Springer, 464–475.
- ALPERN, B., HOOVER, R., ROSEN, B. K., SWEENEY, P. F., AND ZADECK, F. K. 1990. Incremental evaluation of computational circuits. In *Proceedings of the 1st annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 32–42.
- BARAK, A. B. AND ERDŐS, P. 1984. On the maximal number of strongly independent vertices in a random acyclic directed graph. *SIAM J. on Algebraic and Discrete Methods* 5, 4, 508–514.
- DIETZ, P. AND SLEATOR, D. 1987. Two algorithms for maintaining order in a list. *Proceedings of the 19th annual ACM Symposium on Theory of Computing (STOC)*, 365–372.
- ERDŐS, P. AND RÉNYI, A. 1959. On random graphs. *Publ Math Debrecen* 6, 290–297.
- GUIBAS, L. J. AND SEDGEWICK, R. 1978. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 8–21.

- KATRIEL, I. AND BODLAENDER, H. L. 2006. Online topological ordering. *ACM Trans. Algorithms* 2, 3, 364–379. Announced at SODA '05.
- KNUTH, D. E. AND SZWARCFITER, J. L. 1974. A structured program to generate all topological sorting arrangements. *Inf. Process. Lett.* 2, 6, 153–157.
- LIU, H.-F. AND CHAO, K.-M. 2007. A tight analysis of the Katriel-Bodlaender algorithm for online topological ordering. *Theor. Comput. Sci.* 389, 1-2, 182–189.
- MARCHETTI-SPACCAMELA, A., NANNI, U., AND ROHNERT, H. 1993. On-line graph algorithms for incremental compilation. In *Proceedings of International Workshop on Graph-Theoretic Concepts in Computer Science (WG)*. Lecture Notes in Computer Science, vol. 790. 70–86.
- MARCHETTI-SPACCAMELA, A., NANNI, U., AND ROHNERT, H. 1996. Maintaining a topological order under edge insertions. *Information Processing Letters* 59, 1, 53–58.
- OMOHUNDRO, S. M., LIM, C.-C., AND BILMES, J. 1992. The Sather language compiler/debugger implementation. *Technical Report TR-92-017, International Computer Science Institute, Berkeley*.
- ÖSTLIN, A. AND PAGH, R. 2003. Uniform hashing in constant time and linear space. In *Proceedings of the 35th Symposium on Theory of Computing (STOC)*. ACM, 622–628.
- PEARCE, D. J. AND KELLY, P. H. J. 2006. A dynamic topological sort algorithm for directed acyclic graphs. *J. Exp. Algorithmics* 11, 1.7. Announced at WEA '04.
- PEARCE, D. J., KELLY, P. H. J., AND HANKIN, C. 2003. Online cycle detection and difference propagation for pointer analysis. In *Proceedings of the 3rd international IEEE Workshop on Source Code Analysis and Manipulation (SCAM)*.
- RAMALINGAM, G. AND REPS, T. W. 1994. On competitive on-line algorithms for the dynamic priority-ordering problem. *Information Processing Letters* 51, 155–161.
- TARJAN, R. E. 1972. Depth-first search and linear graph algorithms. *SIAM J. Comput.* 1, 2, 146–160.