A Dynamic Topological Sort Algorithm for Directed Acyclic Graphs

DAVID J. PEARCE
Victoria University of Wellington
and
PAUL H. J. KELLY
Imperial College London

We consider the problem of maintaining the topological order of a directed acyclic graph (DAG) in the presence of edge insertions and deletions. We present a new algorithm and, although this has inferior time complexity compared with the best previously known result, we find that its simplicity leads to better performance in practice. In addition, we provide an empirical comparison against the three main alternatives over a large number of random DAGs. The results show our algorithm is the best for sparse digraphs and only a constant factor slower than the best on dense digraphs.

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

General Terms: Algorithms, Experimentation, Theory

Additional Key Words and Phrases: Dynamic graph algorithms, topological sort

1. INTRODUCTION

A topological ordering, ord_D, of a directed acyclic graph $D = (V, E)$ maps each vertex to a priority value such that $ord_D(x) < ord_D(y)$ holds for all edges

This work was supported by the UK Engineering and Physical Sciences Research Council (EPSRC) through a PhD studentship and grant number GR/R15566.
A preliminary version of this paper was presented at the *3rd International Workshop on Efficient and Experimental Algorithms (WEA04)*, 2004. This work includes a more detailed analysis of algorithms for the dynamic topological order problem and an improved experimental study. We must also report that, since the WEA04 workshop, an error in our implementation of algorithm AHRSZ was found. As a result, the performance data reported for AHRSZ in this paper differs from that shown in the workshop paper.
Authors' Addresses: David J. Pearce, School of Mathematics, Statistics and Computer Science, Victoria University of Wellington, Wellington, New Zealand; email: david.pearce@mcs.vuw.ac.nz; Paul H.J. Kelly, Department of Computing, 180 Queen's Gate, South Kensington Campus, Imperial College London, SW7 2AZ, United Kingdom; email: p.kelly@imperial.ac.uk.

$x \to y \in E$. There exist well-known linear time algorithms for computing the topological order of a DAG (e.g., Cormen et al. [2001]). However, these solutions are considered *static* as they compute the solution from scratch.

In this paper, we examine efficient algorithms for updating the topological order of a DAG after some graph change (e.g., edge insertion); we refer to this as the *dynamic topological order (DTO)* problem. We say that an algorithm is *fully dynamic* if it supports both edge insertions and deletions. A partially dynamic algorithm is termed *incremental/decremental* if it supports only edge insertions/deletions. Furthermore, an algorithm is described as *unit change* if it offers no advantage to processing updates in batches rather than one at a time. The main contributions of this paper are as follows:

1. A new fully dynamic, unit change algorithm for maintaining the topological order of a directed acyclic graph.
2. The first experimental study of algorithms for this problem. We compare against two previous algorithms [Marchetti–Spaccamela et al. 1996; Alpern et al. 1990] and a simple, asymptotically optimal static solution.

We show that, while our algorithm has inferior time complexity compared with [Alpern et al. 1990], its simplicity leads to better overall performance, in practice. We also find that our algorithm is significantly more efficient than that of Marchetti–Spaccamela et al. on sparse digraphs and only a constant factor slower on dense digraphs.

The rest of this paper is organized as follows: Section 2 covers necessary background material; Section 3 begins with the presentation of our new algorithm, followed by a detailed discussion of the two main previous solutions [Marchetti–Spaccamela et al. 1996; Alpern et al. 1990]; Section 4 reports on experiments comparing the performance of the three algorithms and the standard (i.e., static) solution using randomly generated digraphs; Section 5 covers related work; finally, we summarize our findings and discuss future work in Section 6.

2. BACKGROUND

At this point, it is necessary to clarify some notation used throughout the remainder of the paper. In the following we assume $D = (V, E)$ is a digraph:

Definition 2.1. We say that x *reaches* a vertex y, written $x \rightsquigarrow y$, if $x = y$ or $x \to y \in E$ or $\exists z.[x \to z \in E \wedge z \rightsquigarrow y]$. We also say that y is reachable from x.

Definition 2.2. The *set of outedges* for a vertex set, $S \subseteq V$, is defined as $E^+(S) = \{x \to y \mid x \to y \in E \wedge x \in S\}$. The *set of inedges*, $E^-(S)$ is defined analogously and the *set of all* edges is $E(S) = E^+(S) \cup E^-(S)$.

Definition 2.3. The extended size of a set of vertices, $K \subseteq V$, is denoted $\|K\| = |K| + |E(K)|$. This definition originates from Alpern et al. [1990].

For the standard (i.e., static) topological sorting problem, algorithms with $\Theta(\|V\|)$ (i.e., $\Theta(v+e)$) time are well known (e.g., Cormen et al. [2001]). However, the problem of dynamically maintaining a topological ordering appears to have received little attention. A trivial solution, based upon a standard (i.e., static)

```
procedure add_edges($B$) // $B$ is a batch of updates
  if $\exists x \rightarrow y \in B.[ord(y) < ord(x)]$ then
    perform standard (i.e., static) topological sort
```

Fig. 1. Algorithm STO, a simple solution to the DTO problem, where *ord* is implemented as an array of size $|V|$.

topological sort, is shown in Figure 1. Note that we generally omit the D from ord_D when it is clear from the context. This algorithm implements *ord* using an array of size $|V|$, which maps each vertex to a unique integer from $\{1 \ldots |V|\}$. Thus, *ord* is a total and contiguous ordering of vertices. The idea is to perform a full topological sort only when an edge $x \rightarrow y$ is inserted, which breaks the ordering (i.e., when $ord(y) < ord(x)$). Therefore, STO traverses the entire graph for one-half of all possible edge insertions and, for a single edge insertion, has a lower and upper bound on its time complexity of $\Omega(1)$ and $O(\|V\|)$, respectively. An important observation is that edge deletions are trivial, since they cannot invalidate the ordering.

In practice, STO performs poorly unless the batch size is sufficiently large and several works have attempted to improve upon it [Alpern et al. 1990; Marchetti–Spaccamela et al. 1996; Hoover 1987; Zhou and Müller 2003; Ramalingam and Reps 1994]. Of these, only two are of interest, since they provide the key results in this field. Henceforth, they are referred to as AHRSZ [Alpern et al. 1990] and MNR [Marchetti–Spaccamela et al. 1996]. We examine these algorithms in some detail later on, but, first, we consider the known results on their time complexity. MNR has been shown to require $O(ve)$ time to process any sequence of $\Theta(e)$ edge insertions [Marchetti–Spaccamela et al. 1996]. One difficulty with this result is that it does not tell us whether the algorithm is (in any sense) optimal. To that end, the work of Alpern et al. is more enlightening as they used an alternative mechanism for theoretically evaluating their algorithm. Their approach was to develop a complexity parameter capturing the *minimal* amount of work needed to update a topological order:

Definition 2.4. Let $D = (V, E)$ be a directed acyclic graph and *ord* a valid topological order. For an edge insertion, $x \rightarrow y$, the set K of vertices is a cover if $\forall a, b \in V.[a \rightsquigarrow b \wedge ord(b) < ord(a) \Rightarrow a \in K \vee b \in K]$.

This states that, for any a and b connected by some path, which are incorrectly prioritized, a cover K must include a or b or both. We say a cover is minimal, written K_{min}, if it is not larger than any valid cover. Thus, K_{min} captures the least number of vertices any algorithm must reorder to obtain a solution. Alpern et al. recognized it that is difficult to do this without traversing edges adjacent to those being reordered. They used a variation on this parameter, which we call K^*_{min}, where $\|K^*_{min}\| \leq \|K\|$ for any valid cover K. Therefore, $\|K^*_{min}\|$ captures the minimal amount of work required, *assuming adjacent edges must be traversed*. It remains an open problem as to whether this assumption is true of all algorithms for this problem. Certainly, it holds for those being studied here.[1] Algorithm AHRSZ obtains an $O(\|K^*_{min}\| \log \|K^*_{min}\|)$ bound on the time

[1]Strictly speaking, only if a refined notion of extended size (see Definition 2.8) is used.

required for a single edge insertion. In contrast, we show in Section 3.2 that MNR is not bounded by $\|K^*_{min}\|$ and, thus, that it has inferior time-complexity.

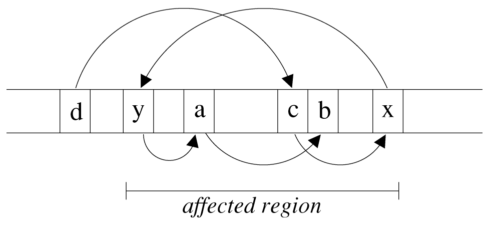
2.1 The Complexity Parameter δ_{xy}

In the above, we introduced the complexity parameter $\|K^*_{min}\|$ as a measure of the least work an algorithm must perform to update an invalidated topological order. Unfortunately, the time complexity for most of the algorithms examined in this paper cannot be expressed in terms of $\|K^*_{min}\|$. Therefore, we must use an alternative to evaluate and understand them:

Definition 2.5. Let $D = (V, E)$ be a directed acyclic graph and *ord* a valid topological order. For an edge insertion $x \to y$, the affected region is denoted AR_{xy} and defined as $\{k \in V \mid ord(y) \leq ord(k) \leq ord(x)\}$.

Definition 2.6. Let $D = (V, E)$ be a directed acyclic graph and *ord* a valid topological order. For an edge insertion $x \to y$, the set δ_{xy} is defined as $\delta^F_{xy} \cup \delta^B_{xy}$, where $\delta^F_{xy} = \{k \in AR_{xy} \mid y \rightsquigarrow k\}$ and $\delta^B_{xy} = \{k \in AR_{xy} \mid k \rightsquigarrow x\}$.

Notice that, $\delta_{xy} = \emptyset$ only when x and y are already correctly prioritized (i.e., when $ord(x) < ord(y)$). Also, it is fairly easy to see that no member of δ^F_{xy} reaches any in δ^B_{xy}, since this would introduce a cycle. To understand δ_{xy} better, it is useful to consider its meaning in a graphical manner:



Here, vertices are laid out in topological order (i.e., increasing in *ord* value) from left to right and the gaps may contain vertices, which we have omitted to simplify the presentation. The edge $x \to y$ invalidates the topological order (i.e., it has just been inserted) and is referred to as an *invalidating edge*, since $ord(y) < ord(x)$. Thus, $\delta_{xy} = \{y, a, b, c, x\}$, since it must include all those vertices in the affected region, which reach x or are reachable from y. One feature common to all the algorithms we will consider is that they only reorder vertices *within the affected region*. This is possible because, for any edge $v \to w$ where $v \notin AR_{xy}$ and $w \in AR_{xy}$, we can reposition w anywhere within the affected region without breaking the invariant $ord(v) < ord(w)$. A similar argument holds when $v \in AR_{xy}$ and $w \notin AR_{xy}$. Another interesting property is the following:

LEMMA 2.7. *Let $D = (V, E)$ be a directed acyclic graph and ord a valid topological order. For an edge insertion $x \to y$, it holds that $K_{min} \subseteq \delta_{xy}$.*

PROOF. Suppose this were not the case. Then there must be a vertex $v \in K_{min}$, where $v \notin \delta_{xy}$. By Definition 2.4, v is incorrectly prioritized with respect to some vertex w. Thus, either $w \rightsquigarrow v$ or $v \rightsquigarrow w$. Consider the case when $w \rightsquigarrow v$

and, hence, $ord(v) < ord(w)$. Since ord is valid for all edges except $x \rightarrow y$, any path from w to v must cross $x \rightarrow y$. Therefore, $y \rightsquigarrow v$ and $w \rightsquigarrow x$ and we have $v \in AR_{xy}$ as $ord(y) \leq ord(v) \leq ord(w) \leq ord(x)$. A contradiction follows as, by Definition 2.6, $v \in \delta_{xy}$. The case when $v \rightsquigarrow w$ is similar. □

In fact, $K_{min} = \delta_{xy}$ only when both are empty. Now, $\|K^*_{min}\| \leq \|K_{min}\| \leq \|\delta_{xy}\|$ and, hence, we know $\|\delta_{xy}\|$ is not strictly a measure of minimal work for the DTO problem. Finally, it turns out that a refinement on the notion of *extended size* is actually more useful when comparing algorithms for the DTO problem:

Definition 2.8. Let $D = (V, E)$ be a directed acyclic graph and ord a valid topological order. For some set $K \subseteq V$, let the *extended-out size* be $\|K\|^+ = |K| + |E^+(K)|$ and the *extended-in size* be $\|K\|^- = |K| + |E^-(K)|$. Then, for an invalidating edge insertion $x \rightarrow y$, the *total search cost* is $\langle\langle K \rangle\rangle = \|K^F\|^+ + \|K^B\|^-$, where $K^F = \{z \in K \mid y \rightsquigarrow z\}$ and $K^B = \{z \in K \mid z \rightsquigarrow x\}$

Intuitively, the difference between $\|K\|$ and $\langle\langle K \rangle\rangle$ is that the former assumes *all* edges adjacent to a vertex must be iterated, while the latter assumes only *inedges* or *outedges* (not both) need to be. This makes sense as the set of vertices to reorder can always be found by searching *forward* from y and *backward* from x. Furthermore, a forward (backward) search does not traverse the inedges (outedges) of those visited. Finally, in what follows, we often reuse the term K^*_{min} to represent a cover where $\langle\langle K^*_{min} \rangle\rangle \leq \langle\langle K \rangle\rangle$ holds for any valid cover K. While this usage is slightly ambiguous, since a set K which minimizes $\langle\langle K \rangle\rangle$ does not necessarily minimize $\|K\|$, our meaning should always be clear from the context.

3. ALGORITHMS FOR THE DYNAMIC TOPOLOGICAL ORDER PROBLEM

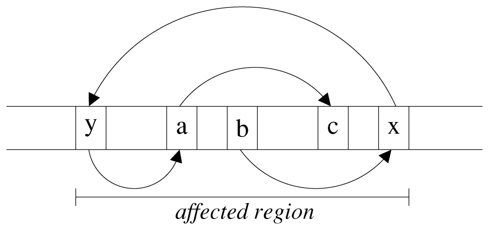
In this section, we begin by presenting our new algorithm for the dynamic topological order problem. We then examine in detail the two main existing solutions and contrast them with our development.

3.1 Algorithm PK

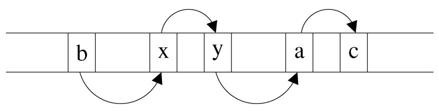
Algorithm PK is our new solution to the DTO problem. As we will see in the coming sections, this is similar in design to MNR, but achieves a time complexity bounded by $\langle\langle \delta_{xy} \rangle\rangle$ (resulting in better performance on sparse graphs). While this still remains inferior to that of AHRSZ, our claim is that its simplicity makes it more efficient in practice. Like all the algorithms under consideration, PK is a unit change algorithm operating on directed acyclic graphs.

The topological ordering, ord, is implemented as a total and contiguous ordering using an array of size $|V|$. This maps each vertex to a unique integer in $\{1 \ldots |V|\}$, such that for any edge $x \rightarrow y$, $ord(x) < ord(y)$ always holds. The main observation behind this algorithm is that we can obtain a correct ordering by simply reorganizing vertices in δ_{xy}. That is, in the new ordering, ord', vertices in δ_{xy} are repositioned to ensure a valid topological ordering, *using only positions previously held by members of* δ_{xy}. All other vertices remain unaffected.

For example:



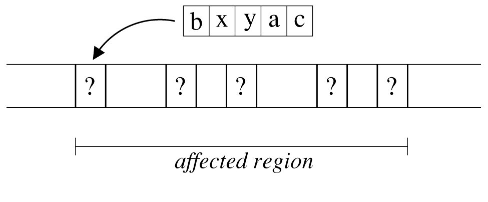
As before, vertices are laid out in topological order from left to right. Only members of δ_{xy} are shown and, as *ord* is total and contiguous, the gaps must contain vertices omitted to simplify the presentation. Thus, $\delta_{xy}^{F} = \{y, a, c\}$ and $\delta_{xy}^{B} = \{b, x\}$ and we obtain a correct ordering by repositioning vertices to ensure all of δ_{xy}^{B} are left of δ_{xy}^{F}:



In doing this, the original (relative) order of vertices in δ_{xy}^{F} must be preserved and likewise for δ_{xy}^{B}. This ensures the following invariant, where ord' is the new ordering being computed, is maintained:

$$\forall x \in \delta_{xy}^{F}.[ord(x) \leq ord'(x)] \wedge \forall y \in \delta_{xy}^{B}.[ord'(y) \leq ord(y)]$$

The above states that members of δ_{xy}^{F} cannot be given lower priorities than they already have, while those in δ_{xy}^{B} cannot get higher ones. This is because, for any vertex in δ_{xy}^{F}, we have identified all in the affected region that must be higher than it (i.e., right of it). However, we have not determined all those which must come lower and, hence, cannot safely move them in this direction. A similar argument holds for δ_{xy}^{B}. Thus, we begin to see how the algorithm works: it first identifies δ_{xy}^{B} and δ_{xy}^{F}. Then, it pools the indices occupied by their vertices and, starting with the lowest, allocates increasing indices first to members of δ_{xy}^{B} and then δ_{xy}^{F}. Thus, in the above example, the algorithm proceeds by allocating b the lowest available index, like so:



After this, it will allocate x to the next lowest index, then y and so on. The algorithm is presented in Figure 2 and the following summarizes the two stages:

```
procedure add_edge($x, y$)
  $lb = ord[y]$;
  $ub = ord[x]$;
  if $lb < ub$ then
    // Discovery
    dfs-f($y$);
    dfs-b($x$);
    // Reassignment
    reorder();

procedure dfs-f($n$)
  $visited(n) = true$;
  $\delta^F_{xy} \cup= \{n\}$;
  forall $n \rightarrow w \in E$ do
    if $ord[w] = ub$ then abort; //cycle
    // is w unvisited and in affected region?
    if $\neg visited(\mathrm{w}) \wedge ord[w] < ub$ then dfs-f($w$);

procedure dfs-b($n$)
  $visited(n) = true$;
  $\delta^B_{xy} \cup= \{n\}$;
  forall $w \rightarrow n \in E$ do
    // is w unvisited and in affected region?
    if $\neg visited(w) \wedge lb < ord[w]$ then dfs-b($w$);

procedure reorder()
  // sort sets to preserve original order of elements
  sort($\delta^B_{xy}$);
  sort($\delta^F_{xy}$);
  $L = \emptyset$;

  // load $\delta^B_{xy}$ onto array L first
  for $i = 0$ to $|\delta^B_{xy}| - 1$ do
    $w = \delta^B_{xy}[i]$;
    $\delta^B_{xy}[i] = ord[w]$;
    $visited(w) = false$;
    push($w, L$);
  // now load $\delta^F_{xy}$ onto array L
  for $i = 0$ to $|\delta^F_{xy}| - 1$ do
    $w = \delta^F_{xy}[i]$;
    $\delta^F_{xy}[i] = ord[w]$;
    $visited(w) = false$;
    push($w, L$);
  merge($\delta^B_{xy}, \delta^F_{xy}, R$);
  // allocate vertices in L starting from lowest
  for $i = 0$ to $|L| - 1$ do $ord[L[i]] = R[i]$;
```

Fig. 2. The PK algorithm. The "sort" function sorts an array such that x comes before y iff $ord[x] < ord[y]$. "merge" combines two arrays into one while maintaining sortedness (i.e., merge sort). Cycles need only be checked for in dfs-f(), since an invalidating edge $x \rightarrow y$ can only give rise to a cycle if y reaches x (and dfs-f() will establish this).

3.1.1 *Discovery.* The set δ_{xy} is identified using a forward depth-first search from y and a backward depth-first search from x. Vertices outside the affected region are not explored. Those visited by the forward and backward search are placed into δ^F_{xy} and δ^B_{xy}, respectively. Thus, $\Theta(\langle\langle\delta_{xy}\rangle\rangle)$ time is needed for this stage. Observe that cycles need only be checked for during the forward search, since an invalidating edge $x \rightarrow y$ can only give rise to a cycle if y reaches x (and the forward search will establish this).

3.1.2 *Reassignment.* The two sets are now sorted separately into increasing topological order (i.e., according to *ord*), which we assume takes $\Theta(\delta_{xy} \log \delta_{xy})$ time. We then load δ^B_{xy} into array L followed by δ^F_{xy}. In addition, the pool of available indices, R, is constructed by merging indices used by elements of δ^B_{xy} and δ^F_{xy} together. Finally, we allocate by giving index $R[i]$ to vertex $L[i]$. This whole procedure takes $\Theta(\delta_{xy} \log\ \delta_{xy})$ time.

Therefore, algorithm PK has time-complexity $\Theta((\delta_{xy} \log\ \delta_{xy}) + \langle\langle\delta_{xy}\rangle\rangle)$. Finally, we provide the correctness proof of algorithm PK:

Lemma 3.1. *Assume $D = (V, E)$ is a DAG and ord an array, mapping vertices to unique values in $\{1 \ldots |V|\}$, which is a valid topological order. If an inserted invalidating edge, $x \rightarrow y$, does not introduce a cycle, then algorithm PK obtains a correct topological ordering.*

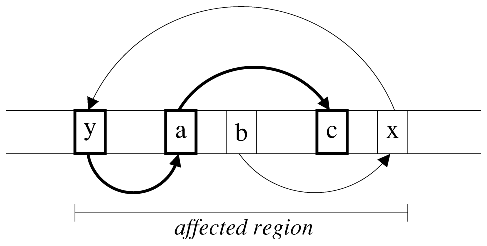
Proof. Let ord' be the new ordering found by the algorithm. To show this is a correct topological order, we must show, for any two vertices a, b where $a \rightarrow b$, that $ord'(a) < ord'(b)$ holds. An important fact to remember is that the algorithm only uses indices of those in δ_{xy} for allocation. Therefore, $z \in \delta_{xy} \Rightarrow ord(y) \leq ord'(z) \leq ord(x)$. There are six cases to consider:

1. $a, b \notin AR_{xy}$. Here neither a nor b have been moved as they lie outside the affected region. Thus, $ord(a) = ord'(a)$ and $ord(b) = ord'(b)$, which (by defn of *ord*) implies $ord'(a) < ord'(b)$.
2. $(a \in AR_{xy} \wedge b \notin AR_{xy}) \vee (a \notin AR_{xy} \wedge b \in AR_{xy})$. When $a \in AR_{xy}$ we know $ord(a) \leq ord(x) < ord(b)$. If $a \in \delta_{xy}$ then $ord'(a) \leq ord(x)$. Otherwise, $ord'(a) = ord(a)$. A similar argument holds when $b \in AR_{xy}$.
3. $a, b \in AR_{xy} \wedge a, b \notin \delta_{xy}$. Similar to case 1 as neither a or b have been moved.
4. $a, b \in \delta_{xy} \wedge x \leadsto a \wedge x \neq a$. Here, a is reachable from x only along $x \rightarrow y$, which means $y \leadsto a \wedge y \leadsto b$. Thus, $a, b \in \delta^F_{xy}$ and their relative order is preserved in ord' by sorting.
5. $a, b \in \delta_{xy} \wedge b \leadsto y \wedge y \neq b$. Here, b reaches y along $x \rightarrow y$, so $b \leadsto x$ and $a \leadsto x$. Therefore, $a, b \in \delta^B_{xy}$ and their relative order is preserved in ord' by sorting.
6. $x = a \wedge y = b$. Here, we have $a \in \delta^B_{xy} \wedge b \in \delta^F_{xy}$ and $ord'(a) < ord'(b)$ follows, because all elements of δ^B_{xy} are allocated lower indices than those of δ^F_{xy}. □

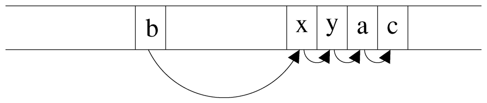
3.2 The MNR Algorithm

The algorithm of Marchetti-Spaccamela et al. also implements *ord* as a total, contiguous ordering of vertices using an array of size $|V|$. As with PK, this maps each vertex to a (unique) integer in $\{1 \ldots |V|\}$. In addition, a second array ord^{-1}

of size $|V|$ is used, which is the reverse of *ord*—it maps each index in the order to the corresponding vertex. Hence, both $ord^{-1}(ord(x)) = x$ and $ord(ord^{-1}(i)) = i$ always hold. The algorithm itself is quite similar to PK except that only δ^F_{xy}, rather than all of δ_{xy}, is identified (using a forward depth-first search). Thus, for the example used, previously only y, a, c would be visited:



To obtain a correct solution the algorithm shifts vertices in δ^F_{xy} up the order so that they hold the highest positions within the affected region. For the above example, this gives the following (valid) ordering:



Notice that these vertices always end up alongside x and that, unlike PK, each vertex in the affected region receives a new position. This has achieved a similar effect to PK, as every vertex in δ^B_{xy} now has a lower index than any in δ^F_{xy}. Pseudocode for the algorithm is presented in Figure 3. The time needed for the DFS (discovery) phase is $\Theta(\|\delta^F_{xy}\|^+)$. The reassignment phase (i.e., procedure *shift*) requires $\Theta(AR_{xy})$ time as each element of AR_{xy} is visited. Therefore, we obtain an $\Theta(\|\delta^F_{xy}\|^+ + AR_{xy})$ bound on the time for a single-edge insertion. Note, only an amortized result was given by Marchetti-Spaccamela et al. and we feel this new result provides better insight into MNR's performance (see Section 3.4 for more on this topic).

3.3 The AHRSZ Algorithm

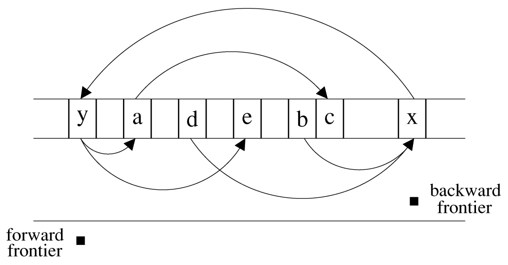
The algorithm of Alpern et al. [1990] employs a special data structure because of Dietz and Sleator to implement a priority space [Dietz and Sleator 1987; Bender et al. 2002]. This permits new priorities to be created between existing ones in $O(1)$ worse-case time. A side effect of using it is that AHRSZ maintains a partial ordering of vertices (unlike PK and MNR, which must maintain a total ordering). Thus, the topological ordering, *ord*, is implemented as an array of size $|V|$, mapping vertices to priority values. Like the others, this algorithm operates in two stages: *discovery* and *reassignment*. We now examine these (assuming $x \to y$ is an invalidating edge):

3.3.1 *Discovery.* The set of vertices, K, to be reprioritized is determined by simultaneously searching forward from y and backward from x. During this,

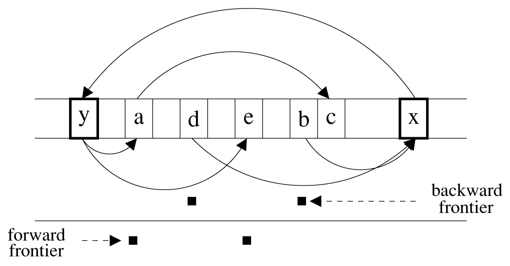
```
procedure add_edge($x, y$)
  $lb = ord[y]$; // lb = lower bound
  $ub = ord[x]$; // ub = upper bound
  if $lb \leq ub$ then
    // invalidating edge
    dfs($y$); // discovery phase
    shift(); // reassignment phase

procedure dfs($n$)
  $visited(n) = true$; // mark n as member of $\delta_{xy}^F$
  forall $n \rightarrow s \in E$ do
    if $ord[s] = ub$ then abort; // cycle detected
    // visit s if not already and is in affected region
    if $\neg visited(s) \wedge ord[s] < ub$ then dfs($s$);

procedure shift()
  $L = \emptyset$;
  $shift = 0$;

  // shift vertices in affected region down ord
  for $i = lb$ to $ub$ do
    $w = ord^{-1}[i]$; // w is vertex at topological index i
    if $visited(w)$ then
      // $w \in \delta_{xy}^F$ so reposition after x
      $visited(w) = false$;
      push($w, L$);
      $shift = shift + 1$;
    else allocate($w, i - shift$);

  // now place members of $\delta_{xy}^F$ in their original order
  for $j = 0$ to $|L| - 1$ do
    allocate($L[j], i - shift$);
    $i = i + 1$;

procedure allocate($n, i$)
  // place n at index i
  $ord[n] = i$;
  $ord^{-1}[i] = n$;
```

Fig. 3. The MNR algorithm.

vertices queued for visitation by the forward (backward) search are said to be on the forward (backward) frontier. At each step, the algorithm extends the frontiers toward each other. The forward (backward) frontier is extended by visiting a member with the lowest (largest) priority. The following diagrams aim to clarify this:



In the above, members of the forward/backward frontiers are marked with a dot. Initially, each frontier consists of a single starting vertex, determined by the invalidating edge. The algorithm proceeds by extending each frontier:



Here we see that the forward frontier has been extended by visiting y and this results in a, e being added and y removed. In the next step, a will be visited as it has the lowest priority of any on the frontier. Likewise, the backward frontier has been extended by visiting x and, next time, b will be visited as it has the *largest* priority. Thus, we see that the two frontiers are moving toward each other and the search stops either when one frontier is empty or they "meet"—when each vertex on the forward frontier has a priority greater than any on the backward frontier. The set of vertices, K, to be reprioritized contains exactly those visited before this happens. We refer to this procedure as *lock-step search*, since both frontiers move in unison.

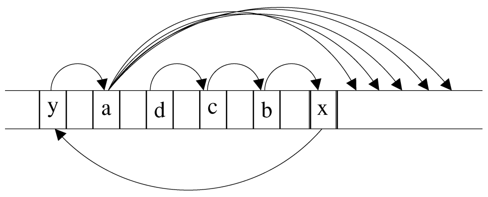
LEMMA 3.2. *Let $D = (V, E)$ be a directed acyclic graph and ord a valid topological order. For an invalidating edge insertion $x \to y$, the set $K \subseteq V$ found by lock-step search is a cover.*

PROOF. Assume it is not. By Definition 2.4, some $a, b \notin K$ exist where $a \leadsto b \wedge ord(a) \geq ord(b)$. Partition K into $K^+ = \{z \in K \mid y \leadsto z\}$ and $K^- = \{z \in K \mid z \leadsto x\}$. Let $F_F = \{w \mid \exists v \in K^+ \wedge v \to w\}$ and $B_F = \{v \mid \exists w \in K^- \wedge v \to w\}$. Now, $\forall v \in F_F, w \in B_F.[ord(v) > ord(w)]$ as the search stops only when this holds. This implies $\forall v \in (\delta^F_{xy} - K^+), w \in (\delta^B_{xy} - K^-).[ord(v) > ord(w)]$, as *ord* is valid for all edges except $x \to y$. The contradiction follows as, by a similar argument to that of Lemma 2.7, $b \in (\delta^F_{xy} - K^+)$ and $a \in (\delta^B_{xy} - K^-)$. □

LEMMA 3.3. *Let $D = (V, E)$ be a directed acyclic graph, ord a valid topological order, and $x \to y$ an invalidating edge insertion. The set $K \subseteq V$ found by lock-step search contains $O(K_{min})$ vertices.*

PROOF. Partition K into $K^+ = \{z \in K \mid y \leadsto z\}$ and $K^- = \{z \in K \mid z \leadsto x\}$. The lock-step search guarantees $|K^+| = |K^-|$ (since both frontiers extend simultaneously) and $\forall v \in K^+, w \in K^-.\big[ord(v) < ord(w)\big]$. Thus, either $K^+ \subseteq K_{min}$ or $K^- \subseteq K_{min}$ must hold, as every vertex in K^+ is incorrectly prioritized with every vertex in K^-. This implies $|K^+| \leq |K_{min}| \leq |K| \leq 2.|K^+| \leq 2.|K_{min}|$. □

Thus, we obtain an $O(\langle\!\langle K_{min} \rangle\!\rangle \log \langle\!\langle K_{min} \rangle\!\rangle)$ bound on discovery using the lock-step search. The log factor arises from the use of priority queues to implement the frontiers, which we assume are heaps. In fact, Alpern et al. use a clever strategy to reduce work further. Consider:



Here, vertex a has high outdegree (which can be imagined as much larger than shown). Thus, visiting vertex a is expensive as its outedges must be iterated. Instead, we could visit b, c, d in potentially much less time and still update the order correctly. The algorithm described so far cannot do this because it moves both frontiers in each step. The full AHRSZ algorithm, however, allows them to move independently to capitalize on situations like the above. Essentially, the frontier whose next vertex has the least number of adjacent edges is moved at each step. If it is a draw, then both are simultaneously moved. Thus, in the above, the backward frontier would be repeatedly extended. To ensure the amount of work done is still strictly bounded by $O(\|K_{min}\|)$, a counter $C(n)$ is maintained for each vertex n. This is initialized by the total number of edges incident on n (i.e., both inedges and outedges). At each step, $min(C(f), C(b))$ is subtracted from $C(f)$ and $C(b)$, where f and b are next on the forward and backward frontiers, respectively. Thus, the forward frontier is extended, if $C(f) = 0$, and the backward, if $C(y) = 0$. Alpern et al. [1990] proved that this ensures an $O(\|K^*_{min}\| \log \|K^*_{min}\|)$ bound on the work done in this stage. This can be improved to $O(\langle\langle K^*_{min}\rangle\rangle \log\langle\langle K^*_{min}\rangle\rangle)$ by initializing $C(n)$ more appropriately [Katriel and Bodlaender 2005]. Specifically, if n is on the forward frontier, then $C(n)$ is initialized with $E^+(n)$, otherwise $E^-(n)$ is used.

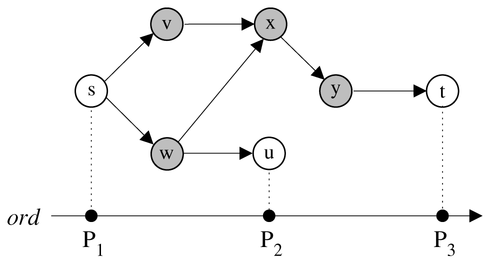
3.3.2 *Reassignment.* The reassignment process also operates in two stages. The first is a depth-first search of all vertices in K, which computes a ceiling on the new priority of each:

$$\begin{aligned} ceiling(x) = min(&\{ord(y) \mid y \notin K \wedge x \to y\} \cup \\ &\{ceiling(y) \mid y \in K \wedge x \to y\} \cup \{+\infty\}) \end{aligned}$$

In a similar fashion, the second stage of reassignment computes the floor using ord', the new topological order formed so far:

$$floor(y) = max(\{ord'(x) \mid x \to y\} \cup \{-\infty\})$$

Once the floor has been computed for a vertex, the algorithm assigns a new priority, $ord'(k)$, such that $floor(k) < ord'(k) < ceiling(k)$. An important consideration here, is to minimize the number of new priorities created [Alpern et al. 1990]. Otherwise, the underlying Dietz and Sleator ordered list structure may not achieve peak performance. Alpern et al. pointed out that, if an arbitrary topological order is used to compute the floor and priority of each $v \in K$, more priorities may be created than necessary. The following example highlights this, where members of K are shaded and the (fixed) priorities of nonmembers are shown below:



The problem is that more priorities are created if v, rather than w, is reassigned first. This is because v must be assigned a priority between it's floor $ord(s)$ and it's ceiling $ord(t)$, *reusing existing priorities whenever possible*. Thus, the new assignment must be $ord(v) = P_2$. This implies each of w, x, and y require a new priority to be created, which is suboptimal, since a valid reassignment is possible that creates only two new priorities. To address this, Alpern et al. use a mechanism similar to breadth-first search to ensure vertices with the same floor get the same priority. Specifically, they employ a min-priority queue with $floor(k)$ as the priority of each member k. Initially, this contains all vertices $k \in K$ with no predecessor in K. The algorithm proceeds by popping all vertices z with the lowest floor off the queue and determining the minimum ceiling, z_{min}, between them. Each z is then assigned the same priority P_z, where $floor(z) < P_z < z_{min}$. In doing this, the lowest existing priority is always used when possible, otherwise a new priority is created. At this point, all remaining vertices whose predecessors are either not in K or have already been reassigned are pushed onto the queue. The whole process is repeated until all of K is reassigned. For the above example, this procedure creates the minimum number of new priorities. However, Alpern et al. did not prove that this holds for the general case, although it seems likely.

Finally, since all edges touching vertices in K must be scanned to generate the floor and ceiling information, the time needed for this stage is bounded by $O(\|K^*_{min}\| + K^*_{min} \log K^*_{min})$. The log factor arises from the use of a min-priority queue. In fact, Katriel and Bodlaender [2005] showed that this can be reduced to $O(K^*_{min})$, using a simpler mechanism. However, this does not minimize the number of new priorities created and, thus, is expected to perform worse in practice.

The original bound given by Alpern et al. on the total time needed to process an edge insertion was $O(\|K^*_{min}\| \log \|K^*_{min}\|)$ [Alpern et al. 1990; Ramalingam and Reps 1994]. This gives $O(\langle\langle K^*_{min}\rangle\rangle \log \langle\langle K^*_{min}\rangle\rangle)$ if the improved discovery algorithm and the simpler approach to reassignment are used. Pseudo-code for our implementation is provided in Figure 4 and there are a few remarks to make about it. In particular, the improved discovery algorithm of Katriel and Bodlaender is used, although their simpler reassignment algorithm is not—*even though it offers lower time complexity*. As discussed above, this is because their approach does not minimize the number of new priorities created and, hence, is expected to perform poorly in practice [Alpern et al. 1990].

```
procedure add_edge($x, y$)
  if $ord(y) < ord(x)$ then $K = \emptyset$; discovery(); reassignment();

procedure discovery()
  $ForwFron = \{y\};\ f = y;\ BackFron = \{x\};\ b = x;$
  $ForwEdges\ =\ OutDegree(f);\ BackEdges\ =\ InDegree(b);$
  // extend frontiers until either one is empty or they meet
  while $ord(f) \leq ord(b)$ do
    $u = min(ForwEdges, BackEdges);$
    $ForwEdges\ =\ ForwEdges - u;$
    $BackEdges\ =\ BackEdges - u;$
    if $ForwEdges = 0$ then
      // extend forward frontier
      $K \cup=\ \{f\};\ ForwFron\ -=\ \{f\};$
      forall $f \rightarrow y \in E$ do $ForwFron \cup= \{y\};$
      if $ForwFron = \emptyset$ then $f = x;$
      else $f\ =\ ForwFron.top();$
      $ForwEdges\ =\ OutDegree(f);$
    if $BackEdges = 0$ then
      // extend backward frontier
      $K \cup=\ \{b\};\ BackFron\ -=\ \{b\};$
      forall $y \rightarrow b \in E$ do $BackFron \cup= \{y\};$
      if $BackFron = \emptyset$ then $b = y;$
      else $b\ =\ BackFron.top();$
      $BackEdges\ =\ InDegree(b);$

procedure reassignment()
  // compute ceilings
  forall $x \in K$ in reverse topological order do
    $ceiling(x) = +\infty;$
    forall $x \rightarrow y \in E$ do
      if $y \in K$ then $ceiling(x) = min(ceiling(y), ceiling(x));$
      else $ceiling(x) = min(ord(y), ceiling(x));$
  // compute new priorities, whilst minimising number created
  $Q = \emptyset;$
  forall $x \in K$ do
    $deps(x) = |\{u \mid u \rightarrow x \wedge u \in K\}|;$
    if $deps(x) = 0$ then $floor(x) = max(\{ord(y) \mid y \rightarrow x \in E\} \cup \{-\infty\});\ Q.push(x);$
  while $Q \neq \emptyset$ do
    $Z = \{z \in Q \mid floor(z) = floor(Q.top())\};$
    $P_z$ = compute_priority($floor(Q.top()), min(\{ceiling(z) \mid z \in Z\})$);
    forall $z \in Z$ do
      $ord(z) = P_z;\ Q.pop();$
      forall $z \rightarrow u \in E$ where $u \in K$ do
        $deps(u) = deps(u) - 1;$
        if $deps(u) = 0$ then $floor(u) = max(\{ord(y) \mid y \rightarrow u \in E\} \cup \{-\infty\});\ Q.push(u);$

procedure compute_priority($floor$, $ceiling$)
  // select lowest priority z where floor < z < ceiling
  // if none exists then create one in O(1) time
  return $z$;
```

Fig. 4. Algorithm AHRSZ. The forward frontier is represented by ForwFron, and implemented using a min-priority queue. BackFron is similar, but using a max-priority queue. Notice that *ForwEdges* and *BackEdges* implement the counter $C(n)$ discussed in the text. Finally, Q is implemented using a min-priority queue.

Finally, there are a few points to make about the Dietz and Sleator [1987] ordered list structure, which AHRSZ relies on: first, it is difficult to implement and suffers high overheads in practice (both in time and space); secondly, only a certain number of priorities can be created for a given word size, thus limiting the maximum number of vertices. For example, only 2^{20} priorities can be created if 32-bit integers are being used.

3.4 Discussion of Complexity

At this point, it seems prudent to clarify the relative complexity of all three algorithms. Comparing AHRSZ and PK is straightforward. The former is bounded by $\langle\langle K^*_{min} \rangle\rangle$ and the latter by $\langle\langle \delta_{xy} \rangle\rangle$. Since $\langle\langle K^*_{min} \rangle\rangle \leq \langle\langle K_{min} \rangle\rangle \leq \langle\langle \delta_{xy} \rangle\rangle$ follows from Lemma 2.7, AHRSZ has a strictly tighter bound on its runtime than PK.

Comparing MNR and PK is more subtle, since neither achieves a strictly tighter bound than the other. Recall that MNR takes $\Theta(\|\delta^F_{xy}\|^+ + AR_{xy})$ time for a single edge insertion, while PK takes $\Theta((\delta_{xy} \log \delta_{xy}) + \langle\langle \delta_{xy} \rangle\rangle)$ time. Furthermore, we have that $\|\delta^F_{xy}\|^+ < \langle\langle \delta_{xy} \rangle\rangle$ and $|\delta_{xy}| \leq |AR_{xy}|$. Thus, if $|AR_{xy}|$ is sufficiently greater than $\langle\langle \delta_{xy} \rangle\rangle$, PK will do less work than MNR (otherwise, the converse is true). Since this is more likely to be true when inserting into a sparse graph, we expect PK to perform better than MNR in these conditions (and viceversa for dense graphs). In practice, however, PK is never much worse than MNR on dense graphs, while MNR can be significantly worse than PK on sparse graphs (as demonstrated in the next section).

3.5 Discussion of Practicality

The central claim of this paper is that, although algorithm PK has an inferior worst-case bound on its runtime compared with AHRSZ, its *simplicity* leads to greater efficiency in practice. The question then, is what it means to be simpler. At a superficial level, it is quite apparent that the pseudocode of AHRSZ (Figure 4) is longer than that of PK (Figure 2). Indeed, while PK is presented in its entirety, all details of the Dietz and Sleator ordered list structure are omitted from the presentation of AHRSZ. A closer inspection of the two algorithms reveals the following observations:

1. During discovery, algorithm PK uses a simple traversal algorithm (i.e., depth-first search) with small overheads; in contrast, AHRSZ uses priority queues (for the frontiers) to implement the traversal (which will almost certainly have higher overheads).
2. During reassignment, algorithm PK sorts the set of discovered vertices (i.e., δ^F_{xy} and δ^B_{xy}) and performs two full passes over them to complete reassignment (note, the *merge* operation is not counted as, in practice, this is done during the last pass at little extra cost); in contrast, AHRSZ performs a topological sort of the discovered set (to compute ceiling information) and then performs two full passes over this set (both of which require iterating adjacent edges) to complete reassignment. Again, it is apparent that AHRSZ must do more work when reassigning a given set of vertices that algorithm PK.

3. Algorithm PK uses an array of integer indices to represent the topological sort, which imposes minimal runtime overhead; AHRSZ, on the other hand, uses the Dietz and Sleator ordered list structure, some of whose operations (such as comparing whether one priority is less than another) can be rather more expensive in practice.

The above gives an informal account of why we believe algorithm PK is simpler than algorithm AHRSZ. In Section 4, we present experimental data, which confirms our hypothesis. We find that, while the number of vertices discovered by AHRSZ is often much less than for PK, the performance of AHRSZ is always worse than PK's—indicating the higher costs involved in its implementation.

4. EXPERIMENTAL STUDY

In this section, we experimentally compare four algorithms for the DTO problem: MNR, AHRSZ, PK, and STO (recall Figure 1). The experiments measure how the *average cost per insertion (ACPI)* varies with *graph density* and batch size, over a large number of randomly generated DAGs.

Definition 4.1. For a DAG with v vertices and e edges, define its density to be $\frac{e}{\frac{1}{2}v(v-1)}$. Thus, it is the ratio of the actual number of edges to the maximum possible.

Furthermore, in an effort to correlate our theoretical analysis, we also investigated how $\langle\langle \delta_{xy} \rangle\rangle$, $|AR_{xy}|$ and $\langle\langle K \rangle\rangle$, where K is the actual cover computed by AHRSZ, vary on average with graph density.

4.1 Generating a Random DAG

The standard model for uniformly generating a random *undirected* graph is $G(v, p)$, which defines a graph with v vertices, where each edge is picked with probability p. Erdös and Rényi [1960] were the first to study this random graph model. They found that, for certain properties such as connectedness, graphs whose edge count was below a certain threshold were very unlikely to have the property, while those with just a few more edges were almost certain to have it. This is known as the *phase transition* and is a curious and pervasive phenomenon (see Janson et al. [2000, Chapter 5] for more on this). Several other random graph models exist, such as one for generating graphs, which obey a power law [Aiello et al. 2000]. For this work, we are only concerned with generating uniform random DAGs and the model $G_{dag}(v, p)$, first defined by Barak and Erdös [1984], is used here:

Definition 4.2. The model $G_{dag}(v, p)$ is a probability space containing all graphs having a vertex set $V = \{1, 2, \ldots, v\}$ and an edge set $E \subseteq \{(i, j) \mid i < j\}$. Each edge of such a graph exists with a probability p independently of the others.

For a DAG in $G_{dag}(v, p)$, we know that there are, at most, $\frac{v(v-1)}{2}$ possible edges. Thus, we can select uniformly from $G_{dag}(v, p)$ by enumerating each possible

```
procedure measure_acpi($v, d, b, s$)
  // v = number of vertices, d = density, b = batch size, s = sample Size
  $ES = \dots$; // generate $d.\frac{1}{2}v(v-1)$ random (acyclic) edges
  $S$ = randomly select $s.\frac{1}{2}v(v-1)$ edges from $ES$;

  $G = (\{1 \dots v\},\ ES-S)$;
  $start$ = timestamp();
  while $S \neq \emptyset$
    $T$ = randomly select $b$ edges from $S$;
    $S = S - T$;
    add_edges($T, G$);

  return $\frac{1}{|S|}$.(timestamp() $-$ $start$);
```

Fig. 5. Our procedure for measuring insertion cost over a random DAG. Note that, through careful implementation, we have minimized the cost of the other operations in the loop, which might have otherwise interfered. In particular, the order in which edges are picked from S is precomputed, using a random shuffle.

edge and inserting with probability p. In our experiments, we used $p = x$ to generate a DAG with v vertices and expected density x.

The approach to generating random DAGs suggested here is by no means the only method. One alternative is to use a *Markov Chain* where each step consists of picking two vertices at random and either deleting the edge between them (if one is present) or inserting an edge between them (if one is not) [Melacon et al. 2001; Ide and Cozman 2002]. Note that, if inserting an edge would introduce a cycle, then nothing is done. In general, it remains unclear how the two generation methods compare and further work could examine this in more detail.

An interesting aspect of our random DAGs is how they are affected by the phase transition phenomenon. This issue was addressed by Pittel and Tungol [2001]. They showed for $G_{dag}(v, p)$ that, if $p = \frac{c(\ln v)}{v}$, then the size of the largest transitive closure of any vertex is asymptotic to $v^c \ln v$, $\frac{2v(\ln \ln v)}{v}$ and $v(1 - \frac{1}{c})$, when $c < 1$, $c = 1$ and $c > 1$, respectively. This means the phase transition occurs roughly at a graph density of $\frac{\ln v}{v}$, after which point it is likely that there exists a vertex connected by a path to every other vertex. In the experiments which follow, the graphs have 2000 vertices and, thus, the phase transition should occur around 0.0038. For this reason, we consider graphs with density below this threshold as sparse, and those over it as dense.

4.2 Experimental Procedure

Our general procedure for measuring the ACPI for an algorithm was to generate, for some $|V|$ and density, a random DAG and measure the time taken to insert a sample of edges while maintaining a topological order. Figure 5 outlines the procedure.[2] Note, the sample size was fixed at 0.0001 (i.e., 0.01% of all $\frac{1}{2}v(v - 1)$ possible edges). Although this seems like a small number, it is

[2]This differs from the procedure used in Pearce and Kelly [2004], which maintained $|E|$ constant during the experiment by deleting edges from within the inner loop. However, we eventually found the overhead of doing this interfered with the results and, thus, we abandoned this approach.

important to realize that most of the interesting observations occur between 0.001 and 0.02 density and, thus, larger sample sizes would swamp our results. To generate each data point, we averaged over 100 runs of this procedure (i.e., over 100 random DAGs). An important aspect of our procedure is that the sample may include noninvalidating edges and these dilute our measurements, since all four algorithms do no work for these cases. Our purpose, however, was to determine what performance can be expected in practice, where it is unlikely all edge insertions will be invalidating.

As mentioned already, some of our experiments measured the average set size of our complexity metrics, instead of ACPI. The procedure for doing this was almost identical to before, except, instead of measuring time, exact values for $\langle\langle K\rangle\rangle$, $\langle\langle \delta_{xy}\rangle\rangle$ and $|AR_{xy}|$ were recorded. These were obtained from the corresponding algorithm (AHRSZ for $\langle\langle K\rangle\rangle$, PK for $\langle\langle \delta_{xy}\rangle\rangle$ and MNR for $|AR_{xy}|$) by counting vertices visited and edges iterated where appropriate.

Finally, all experiments were performed on a 900-Mhz Athlon-based machine with 1 GB of main memory, running Redhat 8.0. The executables were compiled using gcc 3.2, with optimization level "-O3" and timing was performed using the `gettimeofday` function, which gives microsecond resolution. To reduce interference, experiments were performed with all nonessential system daemons/services (e.g., X windows, `crond`) disabled and no other user-level programs running. The implementation itself was in C++ and took the form of an extension to the *Boost Graph Library* [Siek et al. 2002] and utilized the `adjacency_list` class to represent the DAG. Our implementation of AHRSZ employs the $O(1)$ amortized (not $O(1)$ worse-case) time structure of Dietz and Sleator [1987]. This seems reasonable as they themselves state it likely to be more efficient in practice. The complete implementation, including C++ code for all three algorithms and the random graph generator, is available online at `http://www.mcs.vuw.ac.nz/~djp`.

4.3 Single Insertion Experiments

The purpose of these experiments was to investigate the performance of the three unit change algorithms, *AHRSZ*, *PK*, and *MNR*. Specifically, we examined how ACPI varied with $|E|$ and we report our findings here. Furthermore, we include data for a control experiment (labeled as CTRL), whose purpose is to indicate the best possible performance any algorithm could obtain. To generate data for our control, we perform exactly the same steps as for the other algorithms, except that no work is done to actually maintain the topological order. Thus, it measures the cost of inserting edges into our underlying graph data structure.

Figure 6 shows the effect on ACPI and the complexity parameters of varying density, while maintaining $|V|$ constant. Although the highest density shown is 0.1, we have explored beyond this and found the plots extend as expected. Therefore, we limit our attention to this density range as it is most interesting. From the topmost graphs, we see that all three algorithms have quite different behavior. The main observations are: first, MNR performs poorly on sparse graphs, but is the most efficient on dense graphs; second, PK performs well

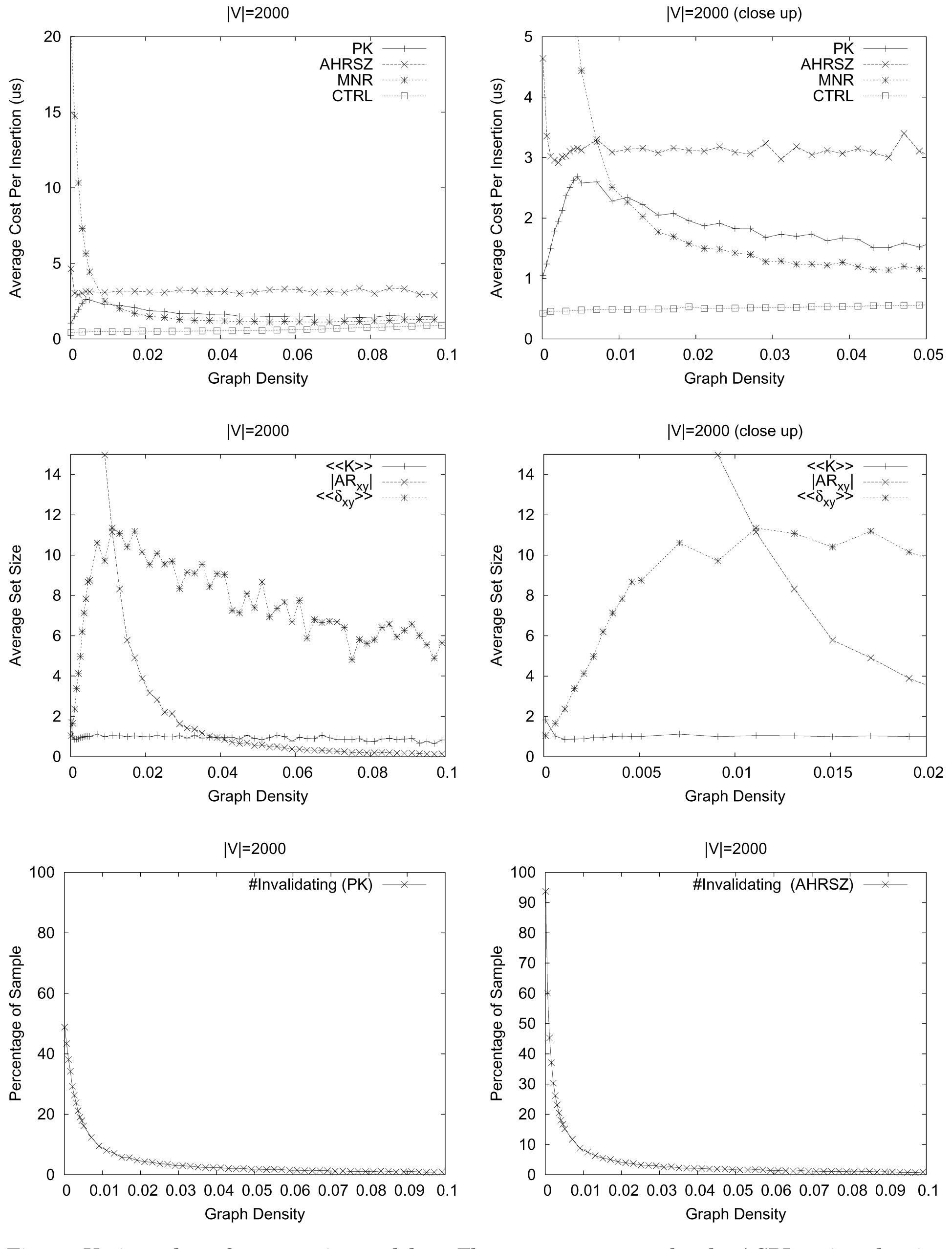


Fig. 6. Various plots of our experimental data. The top-most two graphs plot ACPI against density for the three algorithms and our control. The middle two graphs plot the complexity metrics, which measure the work done by each algorithm. Finally, the bottom two graphs show the percentage of invalidating edges encountered when processing the insertion sample for PK and AHRSZ.

on very sparse and dense graphs, but not as well on those in between; finally, AHRSZ is relatively poor on very sparse graphs, but otherwise has constant performance, which is reasonably competitive with the others. By looking at the middle two graphs of Figure 6, a clear resemblance can be seen between the plots of ACPI for PK and $\langle\langle \delta_{xy} \rangle\rangle$, between that for MNR and $|AR_{xy}|$, and between that for AHRSZ and $\langle\langle K \rangle\rangle$.

The curves observed for the three complexity metrics are key to understanding the performance of the algorithms. Their shape can be explained if we consider the number of invalidating edges in the insertion sample. The bottom two graphs of Figure 6 plot this and they show that the proportion of invalidating edges goes down rapidly with density. However, why is this? Well, we know that as density increases, the chance of a path existing between any two vertices must also increase. From this, it follows that the number of possible invalidating edges must go down as density goes up. This is because an edge $x \rightarrow y$ is invalidating only if there is *no* path from x to y. The steepness of these plots is governed by the phase-transition phenomenon, which dictates that the chance of a path existing between two vertices quickly approaches 1 as soon as the 0.0038 density threshold is passed. From these facts, the curves seen for $|AR_{xy}|$ and $\langle\langle \delta_{xy} \rangle\rangle$ can be explained: first, the average size of an affected region must go down as density increases, since $|AR_{xy}| = 0$ for noninvalidating edges; second, the average size of $\langle\langle \delta_{xy} \rangle\rangle$ must (initially) increase with density, since its size is determined by the chance of a path existing between two vertices. However, the decreasing number of invalidating edges will eventually overpower this and, hence, $\langle\langle \delta_{xy} \rangle\rangle$ is determined by the trade-off between these two factors.

The shape seen for $\langle\langle K \rangle\rangle$ is more subtle. We had expected to see something more closely resembling that of $\langle\langle \delta_{xy} \rangle\rangle$. That is, we had expected to see $\langle\langle K \rangle\rangle$ go up initially and then fall. In fact, a small positive gradient can be seen roughly between 0.001 and 0.005 density which, we argue, corresponds to the increasing chance of a path existing between two vertices at this point. The most important feature of this plot, namely the negative initial gradient, is more curious. In particular, it seems strange that $\langle\langle K \rangle\rangle$ is ever larger than $\langle\langle \delta_{xy} \rangle\rangle$. This does make sense, however, if we contrast the bottom two graphs of Figure 6 against each other. What we see is that the proportion of invalidating edges for AHRSZ starts at a much higher point than for PK. This arises because, on very sparse graphs, AHRSZ will assign most vertices the same priority—so most insertions are invalidating. In contrast, for PK, all vertices have a different priority, regardless of density. This means there is (roughly) a 50% chance that any edge insertion $x \rightarrow y$ will be invalidating, since y is equally likely to come after x in the ordering than before it. Thus, as both $\langle\langle K \rangle\rangle$ and $\langle\langle \delta_{xy} \rangle\rangle$ are empty on valid insertions, we can see that $\langle\langle \delta_{xy} \rangle\rangle$ is smaller than $\langle\langle K \rangle\rangle$ on very sparse graphs simply because it is measured over fewer invalidating edges. Unfortunately, it still remains somewhat unclear why a negative gradient is seen for $\langle\langle K \rangle\rangle$.

Finally, while $\langle\langle K \rangle\rangle$ is generally much smaller than $\langle\langle \delta_{xy} \rangle\rangle$, AHRSZ still performs worse than PK and this reflects the larger constants involved in its implementation (see Section 3.5 for more on this). The fact that MNR outperforms PK on dense graphs is also expected following the discussion of Section 3.4. What may be surprising, however, is that MNR only ever achieves a constant

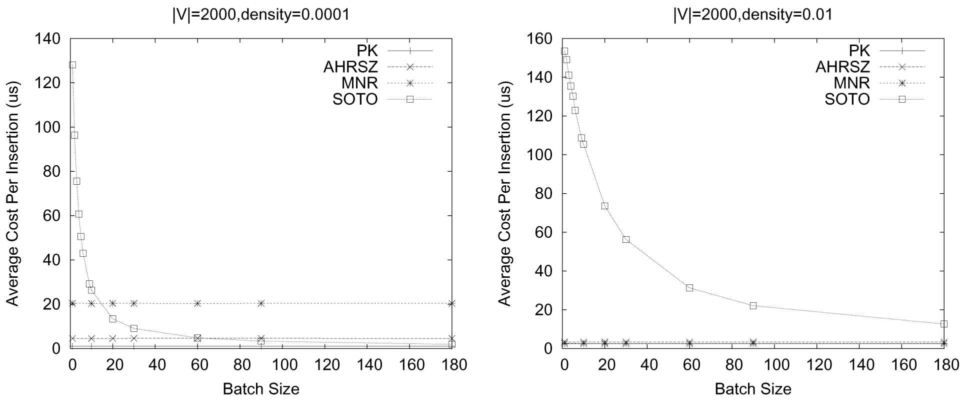


Fig. 7. Plots of our experimental data, illustrating for each algorithm how ACPI varies with batch size for sparse (density 0.0001) and dense graphs (density 0.01).

factor improvement over PK on dense graphs. MNR outperforms PK because it only uses a forward search (i.e., $\|\delta^F_{xy}\|^+$), rather than a forward *and* backward search (i.e., $\langle\!\langle\delta_{xy}\rangle\!\rangle$) as PK does. However, for our random graphs, it is likely that $\|\delta^F_{xy}\|^+ \approx \|\delta^B_{xy}\|^-$ and, hence, only a constant factor improvement is obtained. We suspect this property will hold true for most real-world problems, provided there are sufficient edge insertions. This is because, in a dense graph, it is difficult to construct situations where edge insertions consistently yield significantly smaller forward-search trees compared with backward-search trees.

4.4 Batch Insertion Experiments

The purpose of these experiments was to investigate the performance of the three dynamic algorithms against STO, which you may recall from Figure 1, uses a standard (i.e., static) topological sort based on depth-first search. Thus, following the same experimental procedure as before, we measured ACPI for these algorithms while maintaining $|V|$ and $|E|$ constant and varying the batch size b.

Figure 7 shows the performance of PK, MNR, AHRSZ, and STO across varying batch sizes on sparse and dense graphs. They show a significant advantage is to be gained from using the dynamic algorithms when the batch size is small. Indeed, the data suggests that they compare favorably even for large batch sizes. It is important to realize here that, as the dynamic algorithms are unit change (i.e., they only process one edge at a time), their plots are flat, since they obtain no advantage from seeing the edge insertions in batches. Finally, the plots indicate that, as density increases, the batch size at which STO becomes favorable also increases.

5. RELATED WORK

The main algorithms for the problem of dynamically maintaining a topological order have been covered in Section 3. However, a number of other works on this subject exist. Of particular relevance is that of Katriel and Bodlaender [2005] who obtained an $O(min\{m^{3/2}\log v, m^{3/2} + v^2 \log v\})$ bound to insert

m edges into an empty graph for algorithm AHRSZ. They also showed that, for DAGs with treewidth k, it needs, at most, $O(mk \log^2 v)$ time to insert m edges and that, for the special case of trees, this reduces to $O(v \log v)$. The bound on the time to insert m edges for AHRSZ has since been improved to $O(v^{2.75})$ by Ajwani, Friedrich and Meyer [2006]. Elsewhere, Katriel [2004] has demonstrated algorithm PK to be worse-case optimal with respect to the number of vertices reordered over a series of edge insertions. Zhou and Müller [2003] have improved the space requirements of algorithm AHRSZ. Ramalingam and Reps [1994] proved that no algorithm for the DTO problem can have a constant competitive ratio.

The approach taken in Section 3 to theoretically evaluating algorithms for the DTO problem is known as *incremental complexity analysis*. This methodology is really a natural extension of complexity analysis based on input size. It recognizes that, for a dynamic problem, there is typically no fixed input capturing the minimal amount of work to be performed. Instead, work is measured in terms of a parameter δ representing the (minimal) *change* in input and output required. For example, in the DTO problem, the input is the current DAG and topological order, while the output after an edge insertion is the updated DAG and (any) valid ordering. Thus, δ is the (minimal) set of vertices, which must be reordered (i.e., K_{min}) plus the inserted edge. Incremental complexity analysis is about identifying the parameter δ for the dynamic problem in question. An algorithm is described as *bounded*, if its time complexity can be expressed only in terms of $|\delta|$ or $\|\delta\|$ for all inputs and outputs. Otherwise, it is said to be *unbounded*. The inclusion of $\|\delta\|$ here, as opposed to just $|\delta|$, is simply to include algorithms that depend upon visiting the edges incident to vertices in δ. This is necessary to obtaining a bounded algorithm for all dynamic graph problems we are aware of. The ideas of incremental complexity were developed over several previous works [Berman 1992; Ramalingam and Reps 1996; Ramalingam 1996] and there are many examples of its use (e.g., Reps [1982]; Alpern et al. [1990]; Reps et al. [1986]; Wirn [1993]; Frigioni et al. [1994]; Yeh [1983]; Ramalingam [1996]).

In general, the majority of work on dynamic algorithms for directed graphs has focused on shortest/longest paths and transitive closure (e.g., King and Sagert [1999]; Demetrescu and Italiano [2000]; Djidjev et al. [2000]; Demetrescu et al. [2000]; Frigioni et al. [1998]; Baswana et al. [2002]; Katriel et al. [2005]). For undirected graphs, there has been substantially more work and a survey of this area can be found in Italiano et al. [1999]. Perhaps closest to the problem studied in this paper is that of dynamically identifying strongly connected components in digraphs. We have shown elsewhere how algorithms MNR and PK can be modified for this purpose [Pearce et al. 2004; Pearce 2005].

6. CONCLUSION

We have presented a new algorithm for dynamically maintaining the topological order of a DAG, provided a complexity analysis, correctness proof, and shown that it performs better on sparse graphs than any previously known. Furthermore, we have provided the first experimental comparison of algorithms for this problem over a large number of randomly generated directed acyclic graphs.

For the future, we would like to investigate performance over different classes of random graphs (e.g., power law graphs [Aiello et al. 2000]). We are also aware that random graphs may not reflect real life structures and, thus, experimentation on physically occurring graphs would be useful. Finally, we are particularly interested in finding *batch* variants on these algorithms, which would perform minimal work across a batch of edge insertions.

ACKNOWLEDGMENTS

Special thanks to Irit Katriel for her excellent comments and observations on this work. We also thank Umberto Nanni and the anonymous JEA and WEA referees for their helpful comments on earlier versions of this paper.

REFERENCES

AIELLO, W., CHUNG, F., AND LU, L. 2000. A random graph model for power law graphs. In *Proceedings of the ACM Symposium on the Theory of Computing (STOC)*. 171–180.

AJWANI, D., FRIEDRICH, T., AND MEYER, U. 2006. An $O(n^{2.75})$ algorithm for online topological ordering. In *Proceedings of the Scandinavian Workshop on Algorithm Theory (SWAT)*. Lecture Notes in Computer Science, vol. 4059. Springer-Verlag, New York. 53–74.

ALPERN, B., HOOVER, R., ROSEN, B. K., SWEENEY, P. F., AND ZADECK, F. K. 1990. Incremental evaluation of computational circuits. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*. ACM Press, New York. 32–42.

BARAK, A. AND ERDÖS, P. 1984. On the maximal number of strongly independent vertices in a random acyclic directed graph. *5*, 4, 508–514.

BASWANA, S., HARIHARAN, R., AND SEN, S. 2002. Improved decremental algorithms for maintaining transitive closure and all-pairs shortest paths in digraphs under edge deletions. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*. ACM Press, New York. 117–123.

BENDER, M. A., COLE, R., DEMAINE, E. D., FARACH-COLTON, M., AND ZITO, J. 2002. Two simplified algorithms for maintaining order in a list. In *Proceedings of the European Symposium on Algorithms (ESA)*. Lecture Notes in Computer Science, vol. 2461. Springer-Verlag, New York. 152–164.

BERMAN, A. M. 1992. Lower and upper bounds for incremental algorithms. Ph.D. thesis, Rutgers University, New Brunswick, New Jersey.

CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. 2001. *Introduction to Algorithms*. MIT Press, Cambridge, MA.

DEMETRESCU, C. AND ITALIANO, G. F. 2000. Fully dynamic transitive closure: breaking through the $O(n^2)$ barrier. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*. IEEE Computer Society Press, Washington, DC. 381–389.

DEMETRESCU, C., FRIGIONI, D., MARCHETTI-SPACCAMELA, A., AND NANNI, U. 2000. Maintaining shortest paths in digraphs with arbitrary arc weights: An experimental study. In *Proceedings of the Workshop on Algorithm Engineering (WAE)*. Lecture Notes in Computer Science, vol. 1982. Springer-Verlag, New York. 218–229.

DIETZ, P. F. AND SLEATOR, D. D. 1987. Two algorithms for maintaining order in a list. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*. ACM Press, New York. 365–372.

DJIDJEV, H., PANTZIOU, G. E., AND ZAROLIAGIS, C. D. 2000. Improved algorithms for dynamic shortest paths. *Algorithmica 28*, 4, 367–389.

ERDÖS, P. AND RÉNYI, A. 1960. On the evolution of random graphs. *Mathematical Institute of the Hungarian Academy of Sciences 5*, 17–61.

FRIGIONI, D., MARCHETTI-SPACCAMELA, A., AND NANNI, U. 1994. Incremental algorithms for the single-source shortest path problem. In *Proceedings of the conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*. Lecture Notes in Computer Science, vol. 880. Springer-Verlag, New York. 113–124.

Frigioni, D., Marchetti-Spaccamela, A., and Nanni, U. 1998. Fully dynamic shortest paths and negative cycles detection on digraphs with arbitrary arc weights. In *Proceedings of the European Symposium on Algorithms (ESA)*. Lecture Notes in Computer Science, vol. 1461. Springer-Verlag, New York. 320–331.

Hoover, R. 1987. Ph.D. thesis. Ph.D. thesis, Department of Computer Science, Cornell University, Ithaca, New York.

Ide, J. S. and Cozman, F. G. 2002. Random generation of bayesian networks. In *Proceedings of the Brazillian Symposium on Artificial Intelligence (SBIA)*. Vol. 2507. Springer-Verlag, New York. 366–375.

Italiano, G. F., Eppstein, D., and Galil, Z. 1999. Dynamic graph algorithms. In *Handbook of Algorithms and Theory of Computation, Chapter 22*. CRC Press, Boca Raton, FL.

Janson, S., Luczak, T., and Rucinski, A. 2000. *Random Graphs*. Wiley, New York.

Katriel, I. 2004. Online topological ordering and sorting. Tech. rep., Max-Planck-Institut für Informatik.

Katriel, I. and Bodlaender, H. L. 2005. Online topological ordering. In *Proceedings of the ACM Symposium on Discrete Algorithms (SODA)*. ACM Press, New York. 443–450.

Katriel, I., Michel, L., and Hentenryck, P. V. 2005. Maintaining longest paths incrementally. *Constraints 10*, 2, 159–183.

King, V. and Sagert, G. 1999. A fully dynamic algorithm for maintaining the transitive closure. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*. ACM Press, New York. 492–498.

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

Melaçon, G., Bousquet-Melou, M., and Dutor, I. 2001. Random generation of directed acyclic graphs. In *Proceedings of the Euroconference on Combinatorics, Graph Theory and Applications (COMB)*. Elsevier Science, New York. 12–15.

Pearce, D. J. 2005. Some directed graph algorithms and their application to pointer analysis. Ph.D. thesis, Imperial College, London.

Pearce, D. J. and Kelly, P. H. J. 2004. A dynamic algorithm for topologically sorting directed acyclic graphs. In *Proceedings of the Workshop on Efficient and experimental Algorithms (WEA)*. Lecture Notes in Computer Science, vol. 3059. Springer-Verlag, New York. 383–398.

Pearce, D. J., Kelly, P. H. J., and Hankin, C. 2004. Online cycle detection and difference propagation: Applications to pointer analysis. *Software Quality Journal 12*, 309–335.

Pittel, B. and Tungol, R. 2001. A phase transition phenomenon in a random directed acyclic graph. *RSA: Random Structures & Algorithms 18*, 2, 164–184.

Ramalingam, G. 1996. *Bounded incremental computation*. Ph.D. thesis. Lecture Notes in Computer Science, vol. 1089. Springer-Verlag, New York.

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

Ramalingam, G. and Reps, T. 1996. On the computational complexity of dynamic graph problems. *Theoretical Computer Science 158*, 1–2, 233–277.

Reps, T. 1982. Optimal-time incremental semantic analysis for syntax-directed editors. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. ACM Press, New York. 169–176.

Reps, T., Marceau, C., and Teitelbaum, T. 1986. Remote attribute updating for language-based editors. In *Proceedings of the ACM Symposium on the Principles of Programming Languages (POPL)*. ACM Press, New York. 1–13.

Siek, J., Lee, L.-Q., and Lumsdaine, A. 2002. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, Reading, MA.

Wirn, M. 1993. Bounded incremental parsing. In *Proceedings of the Twente Workshop on Language Technology (TWLT)*. University of Twente, University of Twente, Enschede, The Netherlands. 145–156.

Yeh, D. 1983. On incremental evaluation of ordered attributed grammars. *BIT 23*, 308–320.

Zhou, J. and Müller, M. 2003. Depth-first discovery algorithm for incremental topological sorting of directed acyclic graphs. *Information Processing Letters 88*, 4, 195–200.

Received October 2004; revised March 2006; accepted September 2006