A dynamic algorithm for topologically sorting directed acyclic graphs

David J. Pearce and Paul H. J. Kelly

Department of Computing, Imperial College, London SW7 2BZ, UK
{djp1,phjk}@doc.ic.ac.uk

Abstract. We consider how to maintain 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 marginally 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 three alternatives over a large number of random DAG's. The results show our algorithm is the best for sparse graphs and, surprisingly, that an alternative with poor theoretical complexity performs marginally better on dense graphs.

1 Introduction

A topological ordering, *ord*, of a directed acyclic graph $G = (V, E)$ maps each vertex to a priority value such that, for all edges $x \rightarrow y \in E$, it is the case that $ord(x) < ord(y)$. There exist well known linear time algorithms for computing the topological order of a DAG (see e.g. [4]). However, these solutions are considered *offline* as they compute the solution from scratch.

In this paper we examine *online* algorithms, which only perform work necessary to update the solution after a graph change. We say that an online algorithm is *fully dynamic* if it supports both edge insertions and deletions. The main contributions of this paper are as follows:

1. A new fully dynamic 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 online algorithms [15, 1] and a simple offline solution.

We show that, compared with [1], our algorithm has marginally inferior time complexity, but its simplicity leads to better overall performance in practice. This is mainly because our algorithm does not need the Dietz and Sleator ordered list structure [7]. We also find that, although [15] has the worst theoretical complexity overall, it outperforms the others when the graphs are dense.

Organisation: Section 2 covers related work; Section 3 begins with the presentation of our new algorithm, followed by a detailed discussion of the two previous solutions [1, 15]. Section 4 details our experimental work. This includes a comparison of the three algorithms and the standard offline topological sort; finally, we summarise our findings and discuss future work in Section 5.

2 Related Work

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

Definition 1. *The path relation, $\leadsto$, holds if $\forall x, y \in V.[x \leadsto y \iff x \to y \in E_T]$, where $G_T = (V, E_T)$ is the transitive closure of G. If $x \leadsto y$, we say that x reaches y and that y is reachable from x.*

Definition 2. *The set of edges involving vertices from a set, $S \subseteq V$, is $E(S) = \{x \to y \mid x \to y \in E \wedge (x \in S \vee y \in S)\}$.*

Definition 3. *The extended size of a set of vertices, $K \subseteq V$, is denoted $||K|| = |K| + |E(K)|$. This definition originates from [1].*

The offline topological sorting problem has been widely studied and optimal algorithms with $\Theta(||V||)$ (i.e. $\Theta(|V|+|E|)$) time are known (see e.g. [4]). However, the problem of maintaining a topological ordering online appears to have received little attention. Indeed, there are only two existing algorithms which, henceforth, we refer to as AHRSZ [1] and MNR [15]. We have implemented both and will detail their working in Section 3. For now, we wish merely to examine their theoretical complexity. We begin with results previously obtained:

– AHRSZ - Achieves $O(||\delta|| log ||\delta||)$ time complexity per edge insertion, where δ is the minimal number of nodes that must be reprioritised [1, 19].
– MNR - Here, an amortised time complexity of $O(|V|)$ over $\Theta(|E|)$ insertions has been shown [15].

There is some difficulty in relating these results as they are expressed differently. However, they both suggest that each algorithm has something of a difference between best and worst cases. This, in turn, indicates that a standard worse-case comparison would be of limited value. Determining average-case performance might be better, but is a difficult undertaking.

In an effort to find a simple way of comparing online algorithms the notion of *bounded complexity analysis* has been proposed [20, 1, 3, 19, 18]. Here, cost is measured in terms of a parameter δ, which captures in some way the minimal amount of work needed to update the solution after some incremental change. For example, an algorithm for the online topological order problem will update *ord*, after some edge insertion, to produce a valid ordering *ord'*. Here, δ is viewed as the smallest set of nodes whose priority must change between *ord* and *ord'*. Under this system, an algorithm is described as *bounded* if its worse-case complexity can be expressed purely in terms of δ.

Ramalingam and Reps have also shown that any solution to the online topological ordering problem cannot have a constant competitive ratio [19]. This suggests that competitive analysis may be unsatisfactory in comparing algorithms for this problem.

In general, online algorithms for directed graphs have received scant attention, of which the majority has focused on shortest paths and transitive closure (see e.g. [14, 6, 8, 5, 9, 2]). For undirected graphs, there has been substantially more work and a survey of this area can be found in [11].

3 Online Topological Order

We now examine the three algorithms for the online topological order problem: PK, MNR and AHRSZ. The first being our contribution. Before doing this however, we must first present and discuss our complexity parameter δ_{xy}.

Definition 4. *Let $G = (V, E)$ be a directed acyclic graph and ord a valid topological order. For an edge insertion $x \rightarrow y$, the affected region is denoted AR_{xy} and defined as $\{k \in V \mid ord(y) \leq ord(k) \leq ord(x)\}$.*

Definition 5. *Let $G = (V, E)$ be a directed acyclic graph and ord a valid topological order. For an edge insertion $x \rightarrow y$, the complexity parameter δ_{xy} is defined as $\{k \in AR_{xy} \mid y = k \vee x = k \vee y \rightsquigarrow k \vee k \rightsquigarrow x\}$.*

Notice that δ_{xy} will be empty when x and y are already prioritised correctly (i.e. when $ord(x) < ord(y)$). We say that *invalidating* edge insertions are those which cause $|\delta_{xy}| > 0$. To understand how δ_{xy} compares with δ and the idea of minimal work, we must consider the *minimal cover* (from [1]):

Definition 6. *For a directed acyclic graph $G = (V, E)$ and an invalidated topological order ord, the set K of vertices is a cover if $\forall x, y \in V.[x \rightsquigarrow y \wedge ord(y) < ord(x) \Rightarrow x \in K \vee y \in K]$.*

This states that, for any connected x and y which are incorrectly prioritised, a cover K must include x or y or both. We say that K is minimal, written K_{min}, if it is not larger than any valid cover. Furthermore, we now show that $K_{min} \subseteq \delta_{xy}$:

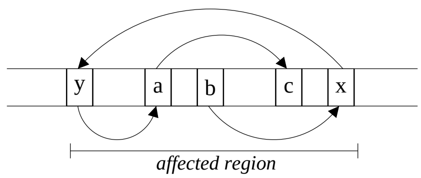
Lemma 1. *Let $G = (V, E)$ be a directed acyclic graph and ord a valid topological order. For an edge insertion $x \rightarrow y$, it holds that $K_{min} \subseteq \delta_{xy}$.*

Proof. Suppose this were not the case. Then a node $a \in K_{min}$, where $a \notin \delta_{xy}$ must be possible. By Definition 6, a is incorrectly prioritised with respect to some node b. Let us assume (for now) that $b \rightsquigarrow a$ and, hence, $ord(a) < ord(b)$. Since ord is valid $\forall e \in E$, except $x \rightarrow y$, any path from b to a must cross $x \rightarrow y$. Therefore, $y \rightsquigarrow a$ and $b \rightsquigarrow x$ and we have $a \in AR_{xy}$ as $ord(y) \leq ord(a) \leq ord(b) \leq ord(x)$. A contradiction follows as, by Definition 5, $a \in \delta_{xy}$. The case when $a \rightsquigarrow b$ is similar.

In fact, $K_{min} = \delta_{xy}$ only when they are both empty. Now, the complexity of AHRSZ is defined in terms K_{min} only and, thus, we know that δ_{xy} is not strictly a measure of minimal work for this problem. Nevertheless, we choose δ_{xy} as it facilitates a meaningful comparison between the algorithms being studied.

3.1 The PK Algorithm

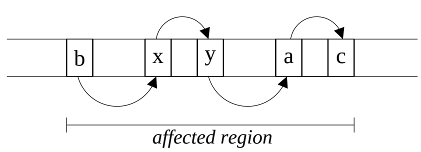
We now present our algorithm for maintaining the topological order of a graph online. As we will see in the coming Sections, it is similar in design to MNR, but achieves a much tighter complexity bound on execution time. For a DAG G, the algorithm implements the topological ordering, *ord*, using an array of size $|V|$, called the *node-to-index* map or $n2i$ for short. This maps each vertex to a unique integer in $\{1 \ldots |V|\}$ such that, for any edge $x \to y$ in G, $n2i[x] < n2i[y]$. Thus, when an invalidating edge insertion $x \to y$ is made, the algorithm must update $n2i$ to preserve the topological order property. The key insight is that we can do this by simply reorganising nodes in δ_{xy}. That is, in the new ordering, $n2i'$, nodes in δ_{xy} are repositioned to ensure a valid ordering, *using only positions previously held by members of* δ_{xy}. All other nodes remain unaffected. Consider the following caused by invalidating edge $x \to y$:



Here, nodes are laid out in topological order (i.e. increasing in $n2i$ value from left to right) with members of δ_{xy} shown. As $n2i$ is a total and contiguous ordering, the gaps must contain nodes, omitted to simplify the discussion. The affected region contains all nodes (including those not shown) between y and x. Now, let us partition the nodes of δ_{xy} into two sets:

Definition 7. *Assume $G = (V, E)$ is a DAG and ord a valid topological order. Let $x \to y$ be an invalidating edge insertion, which does not introduce a cycle. The sets R_F and R_B are defined as $R_F = \{z \in AR_{xy} \mid z = y \vee y \rightsquigarrow z\}$ and $R_B = \{z \in AR_{xy} \mid z = x \vee z \rightsquigarrow x\}$.*

Note, there can be no edge from a member of R_F to any in R_B, otherwise $x \to y$ would a introduce a cycle. Thus, for the above graph, we have $R_F = \{y, a, c\}$ and $R_B = \{b, x\}$. Now, we can obtain a correct ordering by repositioning nodes to ensure all of R_B are left of R_F, giving:



In doing this, the original order of nodes in R_F must be preserved and likewise for R_B. The reason is that a subtle invariant is being maintained:

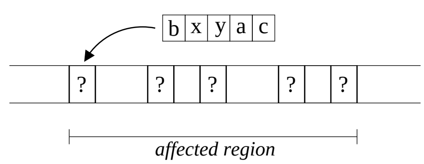
$$\forall x \in R_F . \big[\, n2i[x] \leq n2i'[x] \,\big] \wedge \forall y \in R_B . \big[\, n2i'[y] \leq n2i[y] \,\big]$$

This states that members of R_F cannot be given lower priorities than they already have, whilst those in R_B cannot get higher ones. This is because, for any

```
procedure add_edge(x, y)
  lb = n2i[y]; ub = n2i[x]; if lb < ub then dfs-f(y); dfs-b(x); reassign();

procedure dfs-f(n)
  mark n as visited; R_F ∪= {n};
  forall n→w ∈ E do
   if n2i[w] = ub then abort; //cycle
   if w not visited ∧ n2i[w] < ub then dfs-f(w);

procedure reassign()
  sort(R_B); sort(R_F); L = ∅;
  for i = 0 to |R_B|−1 do w = R_B[i]; R_B[i] = n2i[w]; unmark w; push(w, L);
  for i = 0 to |R_F|−1 do w = R_F[i]; R_F[i] = n2i[w]; unmark w; push(w, L);
  merge(R_B, R_F, R);
  for i = 0 to |L|−1 do n2i[L[i]] = R[i];
```

Fig. 1. The PK algorithm. The "sort" function sorts an array such that x comes before y iff $n2i[x] < n2i[y]$. "merge" combines two arrays into one whilst maintaining sortedness. "dfs-b" is similar to "dfs-f" except it traverses in the reverse direction, loads into R_B and compares against lb. Note, L is a temporary.

node in R_F, we have identified all in the affected region which must be higher (i.e. right) than 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 R_B. Thus, we begin to see how the algorithm works: it first identifies R_B and R_F. Then, it pools the indices occupied by their nodes and, starting with the lowest, allocates increasing indices first to members of R_B and then R_F. So, in the above example, the algorithm proceeds by allocating b the lowest available index, like so:



after this, it will allocate x the next lowest index, then y and so on. The algorithm is presented in Figure 1 and the following summarises the two stages:

Discovery: The set δ_{xy} is identified using a forward depth-first search from y and a backward depth-first search from x. Nodes outside the affected region are not explored. Those visited by the forward and backward search are placed into R_F and R_B respectively. The total time required for this stage is $\Theta(||\delta_{xy}||)$.

Reassignment: The two sets are now sorted separately into increasing topological order (i.e. according to $n2i$), which we assume takes $\Theta(|\delta_{xy}| log\ |\delta_{xy}|)$ time. We then load R_B into array L followed by R_F. In addition, the pool of available indices, R, is constructed by merging indices used by elements of R_B and

R_F together. Finally, we allocate by giving index $R[i]$ to node $L[i]$. This whole procedure takes $\Theta(|\delta_{xy}| log\ |\delta_{xy}|)$ time.

Therefore, algorithm PK has time complexity $\Theta((|\delta_{xy}| log\ |\delta_{xy}|) + ||\delta_{xy}||)$. As we will see, this is a good improvement over MNR, but remains marginally inferior to that for AHRSZ and we return to consider this in Section 3.3. Finally, we provide the correctness proof:

Lemma 2. *Assume $D = (V, E)$ is a DAG and $n2i$ an array, mapping vertices to unique values in $\{1 \ldots |V|\}$, which is a valid topological order. If an edge insertion $x \rightarrow y$ does not introduce a cycle, then algorithm PK obtains a correct topological ordering.*

Proof. Let $n2i'$ 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 $n2i'[a] < n2i'[b]$ holds. An important fact to remember is that the algorithm only uses indices of those in δ_{xy} for allocation. Therefore, $z \in \delta_{xy} \Rightarrow n2i[y] \leq n2i'[z] \leq n2i[x]$. There are six cases to consider:

Case 1: $a, b \notin AR_{xy}$. Here neither a or b have been moved as they lie outside affected region. Thus, $n2i[a] = n2i'[a]$ and $n2i[b] = n2i'[b]$ which (by defn of $n2i$) implies $n2i'[a] < n2i'[b]$.

case 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 $n2i[a] \leq n2i[x] < n2i[b]$. If $a \in \delta_{xy}$ then $n2i'[a] \leq n2i[x]$. Otherwise, $n2i'[a] = n2i[a]$. A similar argument holds when $b \in AR_{xy}$.

Case 3: $a, b \in AR_{xy} \ \wedge\ a, b \notin \delta_{xy}$. Similar to case 1 as neither a or b have been moved.

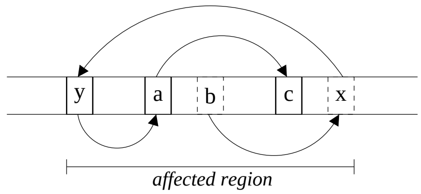
Case 4: $a, b \in \delta_{xy} \ \wedge\ x \leadsto a \ \wedge\ x \neq a$. Here, a reachable from x only along $x \rightarrow y$, which means $y \leadsto a \ \wedge\ y \leadsto b$. Thus, $a, b \in R_F$ and their relative order is preserved in $n2i'$ by sorting.

Case 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, $a, b \in R_B$ and their relative order is preserved in $n2i'$ by sorting.

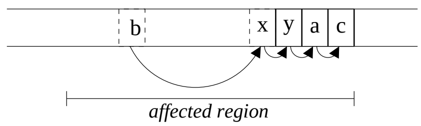
Case 6: $x = a \ \wedge\ y = b$. Here, we have $a \in R_B \wedge b \in R_F$ and $n2i'[a] < n2i'[b]$ follows because all elements of R_B are allocated lower indices than those of R_F.

3.2 The MNR Algorithm

The algorithm of Marchetti-Spaccamela *et al.* operates in a similar way to PK by using a total ordering of vertices. This time two arrays, $n2i$ and $i2n$, of size $|V|$ are used with $n2i$ as before. The second array $i2n$, is the reverse mapping of $n2i$, such that $i2n[n2i[x]] = x$ holds and its purpose is to bound the cost of updating $n2i$. The difference between PK is that only the set R_F is identified, using a forward depth-first search. Thus, for the example we used previously only y, a, c would be visited:



To obtain a correct ordering the algorithm shifts nodes in R_F up the order so that they hold the highest positions within the affected region, like so:



Notice that these nodes always end up alongside x and that, unlike PK, each node in the affected region receives a new position. We can see that this has achieved a similar effect to PK as every node in R_B now has a lower index than any in R_F. For completeness, the algorithm is presented in Figure 2 and the two stages are summarised in the following, assuming an invalidating edge $x \to y$:

Discovery: A depth-first search starting from y and limited to AR_{xy} marks those visited. This requires $O(||\delta_{xy}||)$ time.

Reassignment: Marked nodes are shifted up into the positions immediately after x in $i2n$, with $n2i$ being updated accordingly. This requires $\Theta(|AR_{xy}|)$ time as each node between y and x in $i2n$ is visited.

Thus we obtain, for the first time, the following complexity result for algorithm MNR: $O(||\delta_{xy}|| + |AR_{xy}|)$. This highlights an important difference in the expected behaviour between PK and MNR as the affected region (AR_{xy}) can contain many more nodes than δ_{xy}. Thus, we would expect MNR to perform badly when this is so.

3.3 The AHRSZ Algorithm

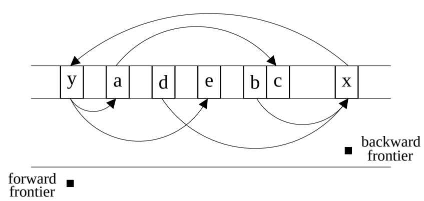
The algorithm of Alpern *et al.* employs a special data structure, due to Dietz and Sleator [7], to implement a priority space which permits new priorities to be created between existing ones in $O(1)$ worse-case time. This is a significant departure from the other two algorithms. Like PK, the algorithm employs a forward and backward search: We now examine each stage in detail, assuming an invalidating edge insertion $x \to y$:

Discovery: The set of nodes, K, to be reprioritised is determined by simultaneously searching forward from y and backward from x. During this, nodes queued for visitation by the forward (backward) search are said to be on the

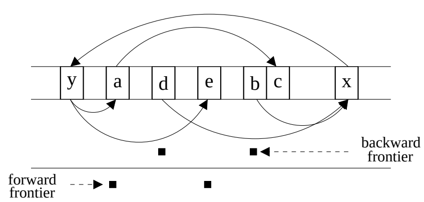
```
procedure add_edge(x, y)
  lb = n2i[y]; ub = n2i[x]; if lb < ub then dfs(y);shift();

procedure dfs(n)
  mark n as visited;
  forall n → s ∈ E do
    if n2i[s] = ub then abort; //cycle
    if s not visited ∧ n2i[s] < ub then dfs(s);

procedure shift()
  L = ∅;
  for i = lb to ub do
    w = i2n[i]; // w is node at topological index i
    if w marked visited then unmark w; push(w, L); shift=shift+1;
    else n2i[L[w]] = i−shift; i2n[i−shift] = w;
  for j = 0 to |L|−1 do
    n2i[L[j]] = i−shift; i2n[i−shift] = L[j]; i=i+1;
```

Fig. 2. The MNR algorithm.

forward (backward) frontier. At each step the algorithm extends the frontiers toward each other. The forward (backward) frontier is extending by visiting a member with the lowest (largest) priority. Consider the following:

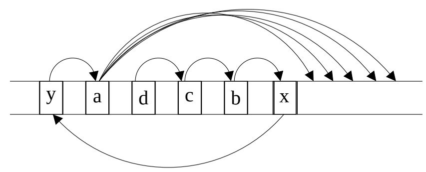


Initially, the frontiers consists of a single starting node, determined by the invalidating edge and the algorithm proceeds by extending each:



Here, members of each frontier are marked with a dot. We see 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 will be extended next time by visiting b 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 node on the forward frontier has a priority greater than any on the backward frontier. An interesting point here is that the frontiers may meet before R_B and R_F have been fully identified. Thus, the discovery stage may identify fewer nodes than that of algorithm PK. In fact, it can be shown that at most $O(|K_{min}|)$ nodes are visited [1], giving an $O(||K_{min}||log||K_{min}||)$ bound on discovery. The *log* factor arises from the use of priority queues to implement the frontiers, which we assume are heaps.

The algorithm also uses another strategy to further minimise work. Consider



where node a has high outdegree (which can be imagined as much larger). Thus, visiting node a is expensive as its outedges must be iterated. Instead, we could visit d, c, b in potentially much less time. Therefore, AHRSZ maintains a counter, $C(n)$, for each node n, initialised by outdegree. Now, let x and y be the nodes to be chosen next on the forward and backward frontiers respectively. Then, the algorithm subtracts $min(C(x), C(y))$ from $C(x)$ and $C(y)$, extending the forward frontier if $C(x) = 0$ and the backward if $C(y) = 0$.

Reassignment: The reassignment process also operates in two stages. The first is a depth-first search of K, those visited during discovery, and computes a ceiling on the new priority for each node, where:

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

In a similar fashion, the second stage of reassignment computes the floor:

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

Note that, $ord'(x)$ is the topological ordering being generated. Once the floor has been computed the algorithm assigns a new priority, $ord'(k)$, such that $floor(k) < ord'(k) < ceiling(k)$. An $O(|K_{min}|log|K_{min}|) + |E(K_{min})|)$ bound on the time for reassignment is obtained. Again, the log factor arises from the use of a priority queue. The bound is slightly better than for discovery as only nodes in K are placed onto this queue.

Therefore, we arrive at a $O(||K_{min}||log||K_{min}||)$ time bound on AHRSZ [1, 19]. Finally, there has been a minor improvement on the storage requirements of AHRSZ [21], although this does not affect our discussion.

3.4 Comparing PK and AHRSZ

We can now see the difference between PK and AHRSZ is that the latter has a tighter complexity bound. However, there are some intriguing differences between them which may offset this. In particular, AHRSZ relies on the Dietz and

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

Fig. 3. Algorithm DFS. Note that *ord* is implemented as an array of size $|V|$.

Sleator ordered list structure [7] and this does not come for free: firstly, 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 nodes. For example, only 32768 priorities (hence nodes) can be created if 32bit integers are being used, although with 64bit integers the limit is a more useful 2^{31} nodes.

4 Experimental Study

To experimentally compare the three algorithms, we measured their performance over a large number of randomly generated DAGs. Specifically, we investigated how insertion cost varies with $|V|$, $|E|$ and batch size. The latter relates to the processing of multiple edges and, although none of the algorithms discussed offer an advantage from this, the standard offline topological sort does. Thus, it is interesting to consider when it becomes economical to use and we have implemented a simple algorithm for this purpose, shown in Figure 3.

To generate a random DAG, we select uniformly from the probability space $G_{dag}(n, p)$, a variation on $G(n, p)$ [12], first suggested in [10]:

Definition 8. *The model $G_{dag}(n, p)$ is a probability space containing all graphs having a vertex set $V = \{1, 2, \ldots, n\}$ 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 $G_{dag}(n, p)$, we know that there are at most $\frac{n(n-1)}{2}$ possible edges. Thus, we can select uniformly from $G_{dag}(n, p)$ by enumerating each possible edge and inserting with probability p. In our experiments, we used $p = \frac{2x}{n-1}$ to generate a DAG with n nodes and expected average outdegree x.

Our procedure for generating a data point was to construct a random DAG and measure the time taken to insert 5000 edges. We present the exact method in Figure 4 and, for each data point, this was repeated 25 times with the average taken. Note that, we wanted the number of insertions measured over to increase proportionally with $|V|$, but this was too expensive in practice. Also, for the batch experiments, we always measured over a multiple of the batch size and chose the least value over 5000. We also recorded the values of our complexity parameters $|\delta_{xy}|, ||\delta_{xy}||$ and $|AR_{xy}|$, in an effort to correlate our theoretical analysis. This was done using the same procedure as before, but instead of measuring time, we traversed the graph on each insertion to determine their values. These were averaged over the total number of edges inserted for 25 runs of the procedure from Figure 4.

Non-invalidating edges were included in all measurements and this dilutes the execution time and parameter counts, since all three algorithms do no work for

procedure measure_acpi(V, E, B, O)
  // measure, in B sized batches, O insertions over a DAG with V
  // nodes and E edges and we assume O = cB, for some c.
  $edgeS$ = gen_random_acyclic_edgeset($V, E + O$);
  $overS$ = randomly select O edges from $edgeS$;

  $G = (\{1 \ldots V\}, edgeS - overS)$;
  $startT$ = timestamp(); // *start timing now*

  while $overS \neq \emptyset$
    T = randomly select B edges from $overS$;
    $overS = overS - T$;
    add_edges(T, G);
    randomly erase B edges from G;

  return (timestamp() $- startT)/O$;

Fig. 4. Our procedure for measuring insertion cost over a random DAG. The algorithm maintains a constant number of edges in G in an effort to eliminate interference caused by varying V, whilst keeping O fixed. Note that, through careful implementation, we have minimised the cost of the other operations in the loop, which might have otherwise interfered. In particular, erasing edges is fast (unlike adding them) and independent of the algorithm being investigated.

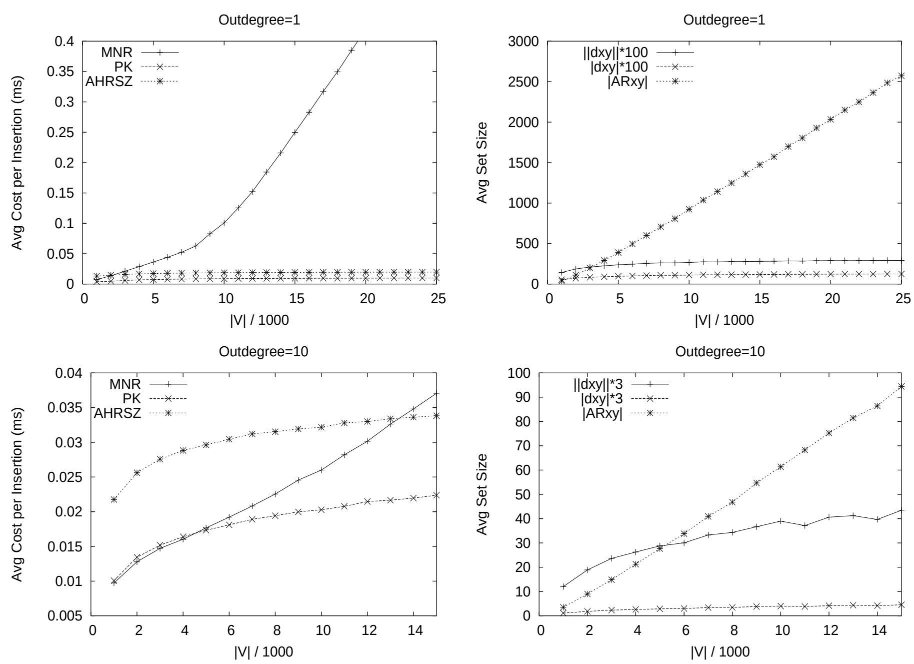


Fig. 5. Experimental data on random graphs with varying $|V|$.

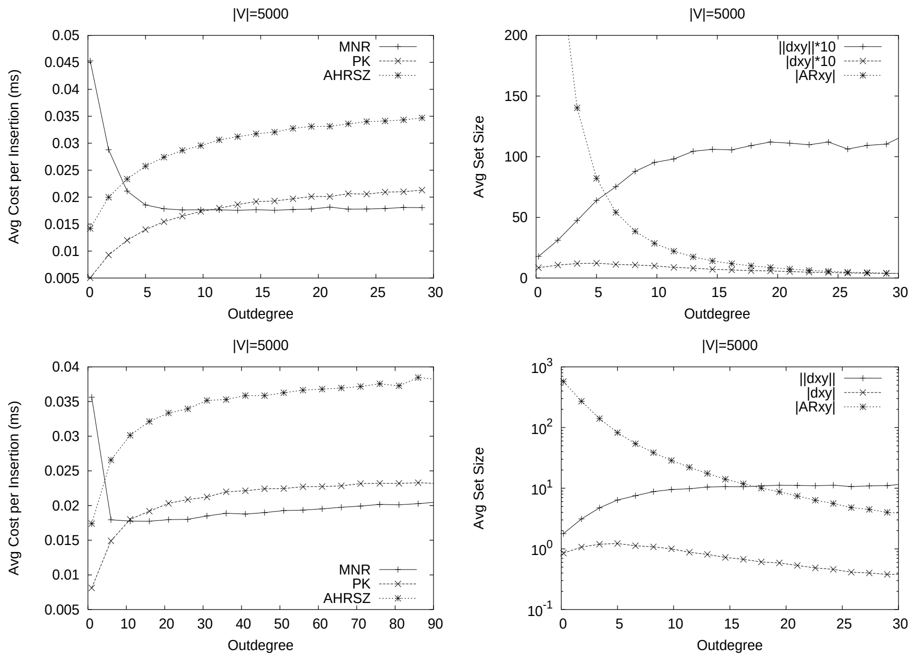


Fig. 6. Experimental data for fixed sized graphs with varying outdegree.

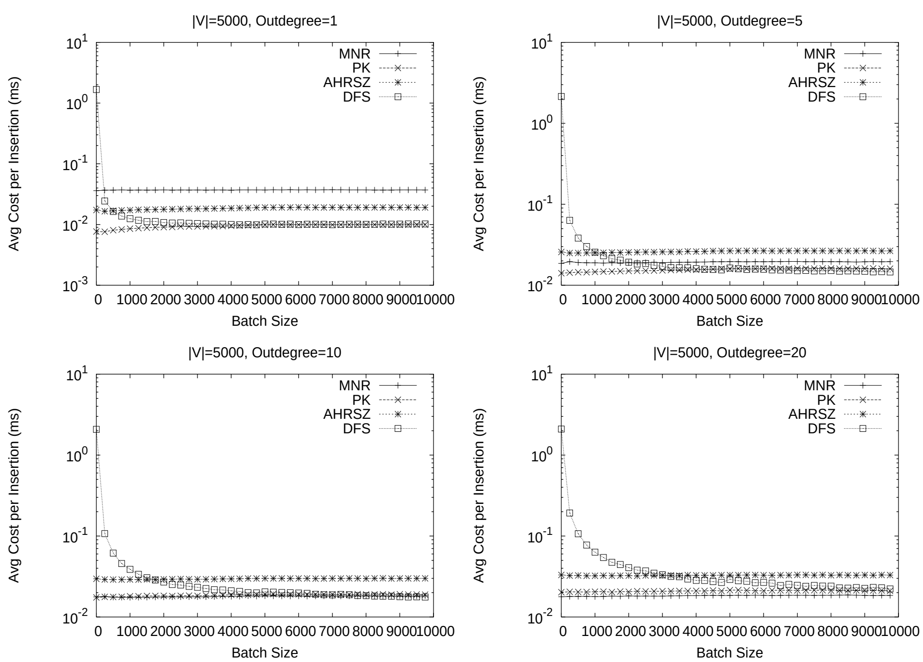


Fig. 7. Experimental data for varying batch sizes comparing the three algorithms against a DFS based offline topological sort

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.

The data, presented in Figures 5, 6 and 7, was generated on a 900Mhz Athlon based machine with 1GB of main memory. Note, we have used some (clearly marked) scaling factors to help bring out features of the data. The implementation itself was in C++ and took the form of an extension to the *Boost Graph Library*. The executables were compiled using gcc 3.2, with optimisation level "-O2" and timing was performed using the `gettimeofday` function. Our implementation of AHRSZ employs the $O(1)$ amortised (not $O(1)$ worse-case) time structure of Dietz and Sleator [7]. This seems reasonable as they themselves state it likely to be more efficient in practice.

4.1 Discussion

The clearest observation from Figures 5 and 6 is that PK and AHRSZ have similar behaviour, while MNR is quite different. This seems surprising as we expected the theoretical differences between PK and AHRSZ to be measured. One plausible explanation is that the uniform nature of our random graphs makes the work saved by AHRSZ (over PK) reasonably constant. Thus, it is outweighed by the gains from simplicity and efficiency offered by PK.

Figure 5: These graphs confirm our theoretical evaluation of MNR, whose observed behaviour strongly relates to that of $|AR_{xy}|$. Furthermore, we expected the average size of AR_{xy} to increase linearly with $|V|$ as $|AR_{xy}| \leq |V|$. Likewise, the graphs for PK and AHRSZ correspond with those of $||\delta_{xy}||$. The curve for $||\delta_{xy}||$ is perhaps the most interesting here. With outdegree 1, it appears to level off and we suspect this would be true at outdegree 10, if larger values of $|V|$ were shown. We know the graphs become sparser when $|V|$ gets larger as, by maintaining constant outdegree, $|E|$ is increasing linearly (not quadratically) with $|V|$. This means, for a fixed sized affected region, $|\delta_{xy}|$ goes down as $|V|$ goes up. However, the average size of the affected region is also going up and, we believe, these two factors cancel each other out after a certain point.

Figure 6: From these graphs, we see that MNR is worst (best) overall for sparse (dense) graphs. Furthermore, the graphs for MNR are interesting as they level off where $|AR_{xy}|$ does not appear to. This is particularly evident from the log plot, where $|AR_{xy}|$ is always decreasing. This makes sense as the complexity of MNR is dependent on both $|AR_{xy}|$ and $||\delta_{xy}||$. So, for low outdegree MNR is dominated by $|AR_{xy}|$, but soon $||\delta_{xy}||$ becomes more significant at which point the behaviour of MNR follows this instead. This is demonstrated most clearly in the graph with high outdegree. Note, when its behaviour matches PK, MNR is always a constant factor faster as it performs one depth-first search and not two. Moving on to $|AR_{xy}|$, if we consider that the probability of a path existing between any two nodes must increase with outdegree, then the chance of inserting an invalidating edge must decrease accordingly. Furthermore, as each non-invalidating edge corresponds to a zero value of $|AR_{xy}|$ in our average, we

can see why $|AR_{xy}|$ goes down with outdegree. Another interesting feature of the data is that we observe both a positive and negative gradient for $|\delta_{xy}|$. Again, this is highlighted in the log graph, although it can be observed in the other. Certainly, we expect $|\delta_{xy}|$ to increase with outdegree, as the average size of the subgraph reachable from y (the head of an invalidating edge) must get larger. Again, this is because the probability of two nodes being connected by some path increases. However, $|\delta_{xy}|$ is also governed by the size of the affected region. Thus, as $|AR_{xy}|$ has a negative gradient we must eventually expect δ_{xy} to do so as well. Certainly, when $|AR_{xy}| \approx |\delta_{xy}|$, this must be the case. In fact, the data suggests the downturn happens some way before this. Note that, although $|\delta_{xy}|$ decreases, the increasing outdegree appears to counterbalance this, as we observe that $||\delta_{xy}||$ does not exhibit a negative gradient. In general, we would have liked to examine even higher outdegrees, but the time required for this has been a prohibitive factor.

Figure 7: These graphs compare the simple algorithm from Figure 3, with the offline topological sort implemented using depth-first search, to those we are studying. They show a significant advantage is to be gained from using the online algorithms when the batch size is small. Indeed, the data suggests that they compare favourably even for sizeable batch sizes. It is important to realise here that, as the online algorithms can only process one edge at a time, their graphs are flat since they obtain no advantage from seeing the edge insertions in batches.

5 Conclusion

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

For the future, we would like to investigate performance over different classes of random graphs (e.g. using the *locality factor* from [10]). We are also aware that random graphs may not reflect real life structures and, thus, experimentation on physically occurring structures would be useful. Another area of interest is the related problem of dynamically identifying strongly connected components and we have shown elsewhere how MNR can be modified for this purpose [17]. We refer the reader to [16], where a more thorough examination of the work in this paper and a number of related issues can be found. Finally, Irit Katriel has since shown that algorithm PK is worse-case optimal, with respect to the number of nodes reordered over a series of edge insertions [13].

Acknowledgements: Special thanks must go to Irit Katriel for her excellent comments and observations on this work. We also thank Umberto Nanni and some anonymous referees for their helpful comments on earlier versions of this paper.

References

1. B. Alpern, R. Hoover, B. K. Rosen, P. F. Sweeney, and F. K. Zadeck. Incremental evaluation of computational circuits. In *Proc. 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 32–42, 1990.
2. S. Baswana, R. Hariharan, and S. Sen. Improved algorithms for maintaining transitive closure and all-pairs shortest paths in digraphs under edge deletions. In *Proc. ACM Symposium on Theory of Computing*, 2002.
3. A. M. Berman. *Lower And Upper Bounds For Incremental Algorithms*. PhD thesis, New Brunswick, New Jersey, 1992.
4. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.
5. C. Demetrescu, D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Maintaining shortest paths in digraphs with arbitrary arc weights: An experimental study. In *Proc. Workshop on Algorithm Engineering*, pages 218–229. LNCS, 2000.
6. C. Demetrescu and G. F. Italiano. Fully dynamic transitive closure: breaking through the $O(n^2)$ barrier. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 381–389, 2000.
7. P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *Proc. ACM Symposium on Theory of Computing*, pages 365–372, 1987.
8. H. Djidjev, G. E. Pantziou, and C. D. Zaroliagis. Improved algorithms for dynamic shortest paths. *Algorithmica*, 28(4):367–389, 2000.
9. D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic shortest paths and negative cycle detection on digraphs with arbitrary arc weights. In *Proc. European Symposium on Algorithms*, pages 320–331, 1998.
10. Y. Ioannidis, R. Ramakrishnan, and L. Winger. Transitive closure algorithms based on graph traversal. *ACM Transactions on Database Systems*, 18(3):512–576, 1993.
11. G. F. Italiano, D. Eppstein, and Z. Galil. Dynamic graph algorithms. In *Algorithms and Theory of Computation Handbook, CRC Press*. 1999.
12. R. M. Karp. The transitive closure of a random digraph. *Random Structures & Algorithms*, 1(1):73–94, 1990.
13. I. Katriel. On algorithms for online topological ordering and sorting. Research Report MPI-I-2004-1-003, Max-Planck-Institut für Informatik, 2004.
14. V. King and G. Sagert. A fully dynamic algorithm for maintaining the transitive closure. In *Proc. ACM Symposium on Theory of Computing*, pages 492–498, 1999.
15. A. Marchetti-Spaccamela, U. Nanni, and H. Rohnert. Maintaining a topological order under edge insertions. *Information Processing Letters*, 59(1):53–58, 1996.
16. D. J. Pearce. *Some directed graph algorithms and their application to pointer analysis (work in progress)*. PhD thesis, Imperial College, London, 2004.
17. D. J. Pearce, P. H. J. Kelly, and C. Hankin. Online cycle detection and difference propagation for pointer analysis. In *Proc. IEEE workshop on Source Code Analysis and Manipulation*, 2003.
18. G. Ramalingam. *Bounded incremental computation*, volume 1089 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
19. G. Ramalingam and T. Reps. On competitive on-line algorithms for the dynamic priority-ordering problem. *Information Processing Letters*, 51(3):155–161, 1994.
20. T. Reps. Optimal-time incremental semantic analysis for syntax-directed editors. In *Proc. Symp. on Principles of Programming Languages*, pages 169–176, 1982.
21. J. Zhou and M. Müller. Depth-first discovery algorithm for incremental topological sorting of directed acyclic graphs. *Information Processing Letters*, 88(4):195–200, 2003.