A Dynamic Batch Algorithm for Maintaining a Topological Order

David J. Pearce  
Computer Science Group  
Victoria University, NZ  
david.pearce@mcs.vuw.ac.nz

Paul H.J. Kelly  
Department of Computing,  
Imperial College, London, UK  
p.kelly@imperial.ac.uk

Abstract

The dynamic topological order problem is that of efficiently updating a topological order after an edge insertion. This can be solved using a standard topological sort algorithm in $O(v + e + b)$ time, for a batch of $b$ edge insertions. However, this always traverses the entire graph when processing a batch of insertions, even if only a few edges are added. Dynamic topological order algorithms traverse only those regions of the graph where the ordering is actually affected by the new edges — usually much less than the whole graph. While these outperform the standard topological sort on relatively small insertion batches, they have a sub-optimal worst-case bound of $O(b(v + e))$. We present, for the first time, a dynamic algorithm with an optimal $O(v + e + b)$ bound on its runtime, which only recomputes those region(s) of the order affected by the new edges. Thus, unlike all other algorithms for this problem, it is efficient on both small and large batches. To support this claim, we empirically evaluate it against the standard topological sort algorithm and a related dynamic topological order algorithm over a large number of random DAGs. The results confirm that our new algorithm is always the best choice.

1 Introduction

A topological order, $ord$, of a directed acyclic graph $D = (V, E)$ maps each vertex to a priority value such that $ord(x) < ord(y)$ holds for all edges $x \rightarrow y \in E$. In this paper, we examine efficient algorithms for updating the topological order of a DAG after an edge insertion and we refer to this as the Dynamic Topological Order (DTO) problem. This problem has received relatively little attention in the past, in spite of having important applications in areas such as program analysis [8]. A simple solution is to use a standard topological sort algorithm to recompute the order from scratch after an edge insertion. For a batch of $b$ edge insertions, this approach has an optimal $O(v + e + b)$ bound on its runtime. In practice, this performs poorly unless the batch size is sufficiently large, as it always visits every vertex and edge when updating the order. In contrast, dynamic topological order algorithms (see [6, 1, 5, 13]) only recompute those region(s) of the order affected by the newly added edges. This means they can visit far fewer vertices and edges when updating the order and, hence, can be significantly faster. However, they need $O(b(v + e))$ time in the worst case to process a batch of $b$ edge insertions, contrasting sharply with the standard topological sort algorithm. We present, for the first time, an algorithm which bridges this gap. Specifically, our contributions are:

(i) The first algorithm for maintaining a topological order which only recomputes those region(s) of the order affected by the newly inserted edges, whilst still obtaining an optimal $O(v + e + b)$ bound on the time needed to update an order after a batch of $b$ edge insertions.

(ii) An experimental study against the standard topological sort algorithm and a related dynamic topological order algorithm confirming our algorithm is always the most efficient (often, significantly so).

2 Background

Pseudo-code for the simple solution to the dynamic topological order problem using a standard topological sort algorithm is shown in Figure 1. This algorithm, henceforth referred to as STS, implements the topological order using an array of size $|V|$ called $ord$. This maps each vertex to a unique integer from $\{1 \ldots |V|\}$, such that $ord(x) < ord(y)$ holds for all edges $x \rightarrow y \in E$. The idea is to perform a full topological sort only when an edge $x \rightarrow y$ is inserted which invalidates the order (i.e. when $ord(y) < ord(x)$). Therefore, STS has a lower and upper bound of $\Omega(1)$ and $O(v + e + b)$ respectively on the time needed to update $ord$ after
a batch of $b$ edge insertions. Note, edge deletions are trivial as they cannot invalidate the order. In practice, STS performs poorly unless the batch size is sufficiently large and several works have attempted to improve upon it [6, 1, 5, 13]. An important idea is the notion of an affected region:

**Definition 1.** Let $D = (V, E)$ 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 $\{ k \in V \mid ord(y) \leq ord(k) \leq ord(x) \}$.

Marchetti-Spaccamela et al. showed that only nodes within the affected region need be repositioned to obtain a valid order [5]. They developed an algorithm, henceforth MNR, based on this observation which processes edges insertions one at a time. Like STS, this implements the topological order using an array of size $|V|$ called $ord$. This maps each vertex to a (unique) integer in $\{1 \ldots |V|\}$ such that $ord(x) < ord(y)$ holds for all edges $x \rightarrow y \in E$. 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. MNR begins by marking nodes reachable from $y$ in the affected region using a depth-first search (known as discovery). Then, it traverses the affected region from bottom-to-top, collecting marked nodes and shifting the rest down (keeping their relative order). This compacts the unmarked nodes at the bottom of the affected region, whilst making space at the top. The marked nodes are then placed into these free slots in their original order. The following illustrates this:

![Image of affected region](image)

The original topological order is shown on the left, while that obtained from running MNR is on the right. In both cases, vertices are laid out in topological order (i.e. increasing in $ord$ value) from left to right. The edge $x \rightarrow y$ invalidates the original order (i.e. it was newly inserted) and is referred to as an invalidating edge, since $ord(y) < ord(x)$. We can see the algorithm has identified those vertices in the affected region reachable from $y$ and has shifted them up the order (i.e. to the right) to obtain a valid ordering. In the worst case, MNR requires $O(v + e)$ time to process an edge insertion. This happens when the affected region includes $O(v)$ nodes and $O(e)$ edges reachable from $y$. To process a batch of $b$ edge insertions, MNR requires $O(\min(b(v+e), v^3))$ time in the worst case because it must process them one at a time. The following illustrates a worst-case scenario for MNR:

**Definition 2.** We say that $x$ reaches a node $y$, written $x \rightsquigarrow y$, if $x = y$ or $x \rightarrow y \in E$ or $\exists z, [x \rightarrow z \in E \land z \rightsquigarrow y]$. We also say that $y$ is reachable from $x$.

![Image of reachability](image)

Here, $n$ is $O(v)$ and there are $O(e)$ existing edges of the form $X_i \rightarrow X_j$, where $i < j$ and $X_1 \rightsquigarrow X_i$. The batch of new insertions contains $b$ edges of the form $Y_i \rightarrow X_1$. If we assume these are processed in order by
MNR, starting from $Y_1 \rightarrow X_1$, then each requires $O(v + e)$ time to complete as every edge reachable from $X_1$ is traversed by the depth-first search. Thus, MNR needs $O(b(v + e))$ time to solve this graph which, in the worst case, is $O(v^3)$. This occurs, for example, when $n = b = \frac{1}{2}v$ and every possible $X_i \rightarrow X_j$ exists. Marchetti-Spaccamela et al. also showed that, even when the batch contains $O(v^2)$ edges (e.g. by allowing those of the more general form $Y_i \rightarrow X_j$ in the above), MNR still takes at most $O(v^3)$ time [5].

3 Algorithm PK

We now present our new algorithm, referred to as PK, for updating the topological order of a DAG after a batch of edge insertions. The algorithm is based upon MNR and, when the batch size is 1, they operate in an identical fashion. As with MNR, algorithm PK employs two arrays, ord and $\text{ord}^{-1}$, to map nodes to indices and vice-versa. To understand how PK works, it is useful to look more carefully at where MNR performs redundant work when processing in batches. This stems from the fact that MNR must process edge insertions one at a time. For example, consider the following, where there are two invalidating edges:

```
  a b c d e f g h i
  \text{affected region 1}
  \text{affected region 2}
```

In the above, $x \rightarrow y$ was chosen first and this led to $y$ and $z$ being visited and shifted passed $x$ (as shown by the middle diagram). Then, $v \rightarrow w$ was processed which resulted in $w, u, x, y$ and $z$ being shifted passed $v$ to obtain the final solution. Thus, we see that $z$ is discovered and shifted twice — once when $x \rightarrow y$ was added and then again for $v \rightarrow w$. Furthermore, every node in both affected regions has also been shifted twice. This is unnecessary since every node which was originally left of and reachable from $v$ had to be shifted right of it. If we could have somehow determined this beforehand then only one shift would have been required. Note, processing $v \rightarrow w$ first does not prevent nodes from being visited and shifted twice.

The key feature of algorithm PK is that it never visits or shifts a node more than once when inserting a batch of edges. In order to achieve this, we must alter our notion of the affected region, which was previously defined as the set of nodes between the head and tail of an invalidating edge. This is done by treating overlapping regions as one — so, although a batch of insertions can still define several affected regions, they are all distinct and can be processed independently. The following aims to clarify this:

```
  e b c d e f g h i
  \text{affected region 1}
  \text{affected region 2}
  \text{affected region 3}
```

Here, each affected region can be correctly ordered independently of the others, by simply rearranging its contents. The difficulty then, lies in rearranging an individual region without visiting or shifting any node twice. This is complicated by the fact that we must now shift nodes to different points within an affected region, instead of only to the rightmost positions (as done in MNR). For example, in the above, $f$ must be positioned just right of $h$, whilst $g$ must go next to $i$. In fact, the reader may wonder why we don’t simply shift both $f$ and $g$ passed $i$. Doing this, it turns out, requires more work as we must also identify and shift any node reachable from $f$ between $h$ and $i$. Therefore, we introduce the shift queue which is a LIFO queue of tuples, $(x, d)$, where $x$ is to be shifted passed $d$ (its destination). For example, the shift queue for processing region 3 above would be: $\{ (f, h), (g, i) \}$. The shifting process operates in much the same way as
that of MNR — by scanning the region from left-to-right whilst filling up vacant slots by moving those not marked during discovery to the left. One difference is that, after moving a node, we check whether it is the destination of any on the shift queue and, if so, place them immediately after it. The following elaboration of region 3 from above shows how it would be shifted using this process:

Here, \( h \) has just been shifted left and, as its destination is reached, \( f \) will be placed into the free slot following it. The algorithm will then proceed by moving \( z \) and \( i \) one slot to the left and placing \( g \). Notice how the queue is carefully arranged so tuples whose destinations will be encountered first are on top. Thus, only a constant-time check is needed to determine whether nodes on the queue need to be placed or not.

The remaining difficulty is the discovery stage which must load the shift queue before each region is reordered. The goal is to ensure each node in the region is visited at most once and the key to achieving this lies in the order with which invalidating edges are processed. Recall the discovery procedure of MNR consists of searching from the head of the invalidating edge to identify and mark those which must be shifted passed its tail. The new procedure remains similar to this, in that we pick an invalidating edge \( x \rightarrow y \) and then search forward from \( y \). This time, however, discovered nodes are placed onto the shift queue, raising the question of what destination to give them. Unfortunately, the obvious answer of using the tail of the invalidating edge does not necessarily work:

Suppose we begin with \( x \rightarrow y \) by searching forward from \( y \) (within the affected region) and adding all encountered to the shift queue. The problem is that the destination for these nodes is actually \( v \), not \( x \), but we cannot know this before processing \( v \rightarrow w \). Therefore, PK processes invalidating edges in decreasing order by the topological index of their tail. In other words, it processes them from right to left. This guarantees it will always start at the rightmost point of any series of connected invalidating edges. Furthermore, if an invalidating edge is traversed (e.g. \( x \rightarrow y \) above) whilst processing another (e.g. \( v \rightarrow w \) above) then it will not be considered again. A subtle point is the way in which the searching is pruned. For MNR, each search was simply restricted to the affected region. However, with our new definition of an affected region this rule leads to some inefficiency:

Here, \( f \) connects to a number of nodes right of \( h \) and, as they lie in the affected region, it seems that a search from \( f \) should visit them. However, it turns out that, since \( h \) is the destination of those discovered from \( f \), only nodes between \( f \) and \( h \) need to be visited. Therefore, algorithm PK restricts the search to just those nodes whose index is lower than the current destination (\( h \) in this case).
are five cases to consider and we now demonstrate how each yields a contradiction of our assumptions: Hence, there must exist two nodes, \( v_\langle Q \rangle \) searching and once whilst shifting. Informally, this follows from the following observations: firstly, on entry to dfs each node is marked visited; secondly, dfs(\( n, \ldots \)) can only be called on unvisited nodes; thirdly, (\( n, \ldots \)) can only be placed on to Q once (at the end of dfs(\( n, \ldots \))); finally, procedure shift allocates each element of Q exactly once and does not complete until Q is empty. In fact, thanks must go to Irit Katriel for pointing out that this can be improved to \( O(v + e + b) \) if a bucket sort is used instead. Thus, PK offers a dramatic improvement over the \( O(\min(b(v + e), v^3)) \) bound obtained by MNR. In fact, all other previously known algorithms for this problem (except STS) also need \( O(v^3) \) in the worst case to process a sequence of edge insertions. Furthermore, although PK does the same amount of work as STS in the worst case, there are many situations when it does considerably less. This is because STS always visits every node and every edge, whereas PK visits only those within an affected region.

**Theorem 1.** Assume \( D = (V, E) \) is a DAG and ord an array mapping each vertex to a unique index from \( \{1 \ldots |V|\} \), with ord\(^{-1} \) implementing its reverse map. If a batch B of edge insertions does not introduce a cycle, then algorithm PK produces a valid topological order.

**Proof.** Let ord’ be the new value of ord computed by the algorithm. Assume ord’ is not a valid topological order. Hence, there must exist two nodes, \( v \) and \( w \), for which \( v \rightarrow w \) and ord’(\( w \)) < ord’(\( v \)) is true. There are five cases to consider and we now demonstrate how each yields a contradiction of our assumptions:

(i) Neither \( v \) nor \( w \) were placed on \( Q \) and ord(\( v \)) < ord(\( w \)). As the variable \( n \) in shift() is only incremented when a member of \( Q \) is reached, any two nodes \( i, j \notin Q \) retain their relative ordering, so ord’(\( v \)) < ord’(\( w \)).

(ii) Neither \( v \) nor \( w \) were placed on \( Q \) and ord(\( v \)) > ord(\( w \)). Since ord is a valid order for all edges except those in \( B \), \( v \) can only reach \( w \) by some series of invalidating edges. Let \( x \rightarrow y \) be the invalidating edge whose tail has the highest priority of any on a path from \( v \) to \( w \). As invalidating edges are sorted into decreasing order by the priority of their tail, it follows that dfs(\( y, \text{ord}(x) \)) is invoked before dfs(\( z, \ldots \)), for any other node \( z \) on a path from \( v \) to \( w \). Furthermore, ord(\( v \)) ≤ ord(\( x \)) must hold — otherwise \( v \) could not reach \( w \) as (by definition of \( x \)) there are no invalidating edges higher up than \( x \). From this and the basic properties of depth-first search it follows that dfs(\( y, \text{ord}(x) \)) invoked dfs(\( w, \text{ord}(x) \)), thus placing \( w \) onto \( Q \) and contradicting our assumption.

(iii) \( \langle w, z \rangle \) was placed on \( Q \), but \( \langle v, \ldots \rangle \) wasn’t. From this, ord(\( w \)) < ord(\( z \)) immediately follows. If ord(\( v \)) ≤ ord(\( z \)) then there is a contradiction as \( w \) is placed (possibly along with other members of \( Q \)) immediately after \( z \). If ord(\( v \)) > ord(\( z \)) then there is an invalidating edge \( x \rightarrow y \) where ord(\( z \)) < ord(\( x \)) and \( x \sim w \), because otherwise \( v \not\sim w \). As invalidating edges are sorted into decreasing order by the priority of their tail, dfs(\( y, \text{ord}(x) \)) must have been invoked before dfs(\( u, \ldots \)), for any other node on a path \( v \sim w \). Again, it follows from the basic properties of depth-first search that dfs(\( y, \text{ord}(x) \)) invoked dfs(\( w, \text{ord}(x) \)). This gives a contradiction, since it implies \( \langle w, x \rangle \) was placed onto the \( Q \) (not \( \langle w, z \rangle \)).

(iv) \( \langle v, z \rangle \) was placed on \( Q \), but \( \langle v, \ldots \rangle \) wasn’t. From this, ord(\( v \)) < ord(\( z \)) immediately follows. If ord(\( w \)) < ord(\( z \)) then dfs(\( v, \text{ord}(z) \)) (which must have been called for \( v \) to be on \( Q \)) lead to dfs(\( w, \text{ord}(z) \)) and \( w \) being pushed onto \( Q \), giving a contradiction. If ord(\( z \)) ≤ ord(\( w \)) then \( \langle z, x \rangle \) was not placed on \( Q \), for any node \( x \). This holds because otherwise \( \langle v, x \rangle \) would have been pushed onto \( Q \) (since \( x \sim v \wedge \text{ord}(z) < \text{ord}(x) \)) and a similar argument to that used in (ii) and (iii) applies. As neither \( z \) nor \( w \) are placed on \( Q \), the argument from (i) gives ord’(\( z \)) < ord’(\( w \)) and a contradiction as it implies \( v \) is placed (possibly with other members of \( Q \)) immediately after \( z \).

(v) Both \( \langle v, x_1 \rangle \) and \( \langle w, x_2 \rangle \) were placed on \( Q \). Again, ord(\( v \)) < ord(\( x_1 \)) ∧ ord(\( w \)) < ord(\( x_2 \)) follows immediately. If it can be shown that \( \langle v, x_1 \rangle \) is pushed onto \( Q \) after \( \langle w, x_2 \rangle \), then a contradiction
procedure add_edges(B)

// remove forward edges from B
forall x→y ∈ B do if ord[x] < ord[y] then B = B − {x→y}
// sort invalidating edges into descending order by index of their tail
sort(B);
Q = ∅;  // the shift queue
lb = |V|;  // lowerbound of current affected region
// work through each affected region in turn processing invalidating edges
for i = 0 . . . |B| do
x→y = B[i];
// if index of tail less than lower bound, current region finished, so shift
if ord[x] < lb then shift(lb);
// dfs from head if edge not already traversed
if ¬visited(y) then dfs(y, ord[x]);
lb = min(ord[y], lb);
// shift final affected region
shift(lb);

procedure dfs(v, ub)
visited(v) = true;
forall v→s ∈ E do
  if ord[s] = ub then abort;  // cycle detected
  if ¬visited(s) ∧ ord[s] < ub then dfs(s, ord[s]);
// place n and current destination in topological sort on queue
push(⟨v, ord−1[ub]⟩, Q)

procedure shift(i)
n = 0;  // number of nodes temporarily removed from order so far
while Q ̸= ∅ do
  w = ord−1[i];  // w is node at topological index i
  if visited[w] then
    visited[w] = false;  // reset visited flag for future calls to add_edge
    n = n + 1;
  else allocate(w, i − n);
// now insert all nodes associated with index i
  ⟨v, t⟩ = top(Q);
  while Q ̸= ∅ ∧ w = t do
    n = n − 1;
    allocate(v, i − n);
    pop(Q);
    ⟨v, t⟩ = top(Q);
    i = i + 1;

procedure allocate(v, i)
// place n at index i
ord[v] = i;  ord−1[i] = v;

Figure 2: Algorithm PK. This first marks those nodes reachable from y in AR_{xy} and then shifts them to lie immediately after x in ord−1. Note that, initially all nodes are marked unvisited. The allocate(v, i) method simply allocates a node v at index i in the order.
follows as nodes are allocated in LIFO order. Thus, it remains only to show this. Let \( x_1 \rightarrow y_1 \) and \( x_2 \rightarrow y_2 \) be the two invalidating edges responsible for pushing \( \langle v, x_1 \rangle \) and \( \langle w, x_2 \rangle \) onto \( Q \) respectively. If \( x_1 = x_2 \) then either \( \text{dfs}(v, \text{ord}(x_1)) \) invoked \( \text{dfs}(w, \text{ord}(x_1)) \) or the latter had already been called (due to some path \( x_1 \rightarrow w \) not involving \( v \)). Either way, \( \langle w, \text{ord}(x_1) \rangle \) is pushed first. If \( x_1 \neq x_2 \) then either \( \text{ord}(x_1) < \text{ord}(x_2) \) and \( \text{dfs}(w, \text{ord}(x_2)) \) was invoked first, or \( \text{ord}(x_1) > \text{ord}(x_2) \) and \( \text{dfs}(v, \text{ord}(x_1)) \) failed to call \( \text{dfs}(w, \text{ord}(x_1)) \). For the latter to hold, it must be that \( \text{ord}(x_1) < \text{ord}(w) \) (otherwise \( w \) was already visited). But, this implies \( \text{ord}(x_1) < \text{ord}(x_2) \) and, hence, that \( \text{dfs}(w, \text{ord}(x_2)) \) was invoked before \( \text{dfs}(v, \text{ord}(x_1)) \). Again, both cases result in \( \langle w, \text{ord}(x_1) \rangle \) being pushed first.

\[ \square \]

4 Experimental Study

In this section, we experimentally compare the performance of algorithm PK against STS and MNR. The experiments measure how the Average Cost Per Insertion (ACPI) varies with batch size at different graph densities, over a large number of randomly generated DAGs.

**Definition 3.** For a DAG with \( v \) nodes and \( e \) edges, define its density to be \( \frac{e}{\binom{v}{2}} \). Thus, it is the ratio of the number of actual edges to the maximum possible.

**Definition 4.** The model \( G_{\text{dag}}(v, p) \) 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.

The model \( G_{\text{dag}}(v, p) \) was first defined by Barak and Erdős [2]. Using this, a DAG with \( v \) nodes and expected density \( x \) can be generated by setting \( p = x \). Our experiments determined the Average Cost Per Insertion (ACPI) for each algorithm by measuring the time taken to insert a sample of edges into a DAG whilst maintaining a topological order. To do this, we generated 100 random DAGs with 2500 vertices at density 0.001, and 100 random DAGs with 2500 vertices at density 0.01. The edge set for each graph was divided into those making up the graph itself and those making up the insertion sample. The size of the insertion sample was fixed at 360 edges to ensure the total amount of work done remained constant across all experiments. For a given algorithm and batch size \( b \), the average time taken to process the insertion sample in batches of \( b \) edges was recorded for each graph. An important point is that the insertion sample may include non-invalidating edges and these dilute our measurements, since the 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.

All experiments were performed on a 900Mhz Athlon based machine with 1GB of main memory, running Mandrake Linux 10.2. The executables were compiled using gcc 3.4.3, with optimisation level “-O3” and timing was performed using the ` gettimeofday` function, which gives microsecond resolution. To reduce interference, experiments were performed with all non-essential 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 v1.33.0, utilising the `adjacency_list` class to represent a DAG [12]. 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](http://www.mcs.vuw.ac.nz/~djp).

Figure 3 shows the results of our experiments comparing ACPI for PK, MNR and STS across varying batch sizes at densities 0.001 and 0.01. The plots for MNR are flat since it processes edges one at a time and, hence, obtains no advantage from larger insertion batches. The main conclusion is that PK is always a better choice than either MNR or STS and, in many cases, offers significant gains. In particular, when the batch size is one, little difference is observed between MNR and PK which reflects their relationship. At density 0.01, the gap between these two algorithms has reduced. The reason for this is that, on dense graphs, there are few invalidating edges in the insertion batch as the graph is already highly ordered (hence, most insertions are not invalidating — see [7] for more on this). Thus, there is less chance the affected regions
Figure 3: Experimental results looking at the effect of increasing batch size for all three algorithms on random DAGs with 2500 nodes at densities 0.001 and 0.01. For each, batch size is plotted against ACPI and we provide close ups at each density to capture interesting features.

Figure 4: Experimental results looking at the effect of increasing batch size for all three algorithms on random DAGs with 2500 nodes at densities 0.001 and 0.01. For each, batch size is plotted against the average number of nodes and edges visited or shifted when processing an edge insertion. We provide close ups at each density to capture interesting features.
for two invalidating edges will overlap (as there are simply fewer affected regions) which is needed for PK to obtain an advantage over MNR. The data also indicates that, for large batches, the performance of STS approaches that of PK. This is expected as it becomes highly likely here that most nodes will be a member of some affected region and, hence, will be reordered by PK anyway. Of course, PK can still obtain an advantage because it doesn’t always need to traverse every edge (as STS does).

Figure 4 shows the results of a second experiment which measured the number of vertices and edges visited or shifted by each algorithm, rather than ACPI (note, all other experimental parameters remain the same). This is useful as it gives us a clear picture regarding the amount of work being performed by each algorithm, which is not muddled by the performance characteristics of the experimental machine. The charts show a striking resemblance to those of Figure 3 and give a strong indication that the results of Figure 3 are not dependent on the experimental machine, rather it is a direct function of the underlying algorithm.

5 Related Work

Aside from MNR and STS, there are two other previously known (unit change) algorithms for this problem. The first, referred to as AHRSZ, is due to Alpern et al. who used a novel complexity analysis to show their algorithm was near-optimal in the case that edges are inserted one at a time (rather than in batches) [1]. Their approach has since become known as incremental complexity analysis and is a natural extension of complexity analysis based on input size. It recognises 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 $\delta$ representing the (minimal) change in input and output required. For example, in the DTO problem, input is the current topological order, while output is (any) valid ordering after an edge insertion. Thus, $\delta$ is the (minimal) set of nodes which must be reordered.

Definition 5. The extended size of a set of vertices, $K \subseteq V$, is denoted $||K|| = |K| + |E(K)|$, where $E(S) = \{x \rightarrow y \in E \mid x \in S \lor y \in S\}$. This definition originates from [1].

An algorithm is described as bounded if its time complexity can be expressed only in terms of $||\delta||$ for all inputs and outputs. Otherwise, it is said to be unbounded. The use of $||\delta||$ here, as opposed to $|\delta|$, is simply to include algorithms which depend upon visiting the edges of nodes in $\delta$. 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 [3, 10, 9] and there are many examples of its use (e.g. [11, 1, 9]). Alpern et al. developed the idea of minimal cover to define $\delta$ for the DTO problem:

Definition 6. 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 $\forall a, b \in V, [a \sim b \land ord(b) < ord(a) \Rightarrow a \in K \lor b \in K]$. This states that for any $a$ and $b$ connected by some path, which are incorrectly prioritised, a cover $K$ must include $a$ or $b$ or both. We say a cover is minimal, written $K_{\text{min}}$, if it is not larger than any valid cover. Thus, $K_{\text{min}}$ captures the least number of nodes any algorithm must reorder to obtain a solution. Alpern et al. recognised it is difficult to do this without traversing edges adjacent to those being reordered. Therefore, they defined $K_{\text{min}}^*$ to be a cover where $||K_{\text{min}}^*|| \leq ||K||$ for any valid cover $K$. Thus, $||K_{\text{min}}^*||$ 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 developed so far. The algorithm of Alpern et al. needs $O(||K_{\text{min}}^*|| \log ||K_{\text{min}}^*||)$ time for a single edge insertion (hence, it is bounded).

Definition 7. Let $D = (V, E)$ be a directed acyclic graph and ord a valid topological order. For an edge insertion $x \rightarrow y$, the set $\delta_{xy}$ is defined as $\delta_{xy}^+ \cup \delta_{xy}^-$, where $\delta_{xy}^+ = \{k \in AR_{xy} \mid y \sim k\}$ and $\delta_{xy}^- = \{k \in AR_{xy} \mid k \sim x\}$. For MNR, $O(||\delta_{xy}^+|| + AR_{xy})$ time is needed to process a single edge insertion [6, 7]. For an invalidating edge $x \rightarrow y$, it holds that $||K_{\text{min}}^*|| \leq ||\delta_{xy}^+||$ and, hence, MNR is unbounded. Elsewhere, we have developed
an algorithm, henceforth PK$_2$, which needs $O(\delta_{xy} \log \delta_{xy} + ||\delta_{xy}||)$ time per insertion [6, 7]. While this is still unbounded, its simplicity and efficiency make it faster than AHRSZ in practice. Note, algorithm PK$_2$ also requires $O(\min(b(v + e), v^3))$ time to process a batch of $b$ edge insertions. Regarding AHRSZ, Katriel and Bodlaender showed an improved version needs $O(\min\{m^{3/2} \log v, m^{3/2} + v^2 \log v\})$ time to insert $m$ edges into an empty graph [4]. This implies that in the worst-case (i.e. when $m = \frac{1}{2}v(v - 1)$) AHRSZ requires $O(\min\{v^3 \log v, v^3 + v^2 \log v\})$ time to process a batch of edge insertions. In contrast, PK only ever needs $O(v^2)$ time.

6 Conclusion

We have presented the first dynamic topological order algorithm which only recomputes those region(s) of the order affected by the newly inserted edges, whilst still obtaining an optimal $O(v + e + b)$ bound on the time needed to update an order after a batch of $b$ edge insertions. We have also shown experimentally that it performs better than a standard topological sort algorithm and the related MNR algorithm on a large number of random graphs.

References


