

A Batch Algorithm for Maintaining a Topological Order

David J. Pearce¹

Paul H.J. Kelly²

¹School of Engineering and Computer Science, Victoria University of Wellington, NZ,
Email: david.pearce@ecs.vuw.ac.nz

²Department of Computing, Imperial College London, UK, Email: p.kelly@imperial.ac.uk

Abstract

The dynamic topological order problem is that of efficiently updating a topological order after some edge(s) are inserted into a graph. Much prior work exists on the unit-change version of this problem, where the order is updated after every single insertion. No previous (non-trivial) algorithms are known for the batch version of the problem, where the order is updated after every batch of insertions. We present the first such algorithm. This requires $O(\min\{k \cdot (v+e), ve\})$ time to process any sequence of k insertion batches. This is achieved by only re-computing those region(s) of the order affected by the inserted edges. In many cases, our algorithm will only traverse small portions of the graph when processing a batch. We empirically evaluate our algorithm against previous algorithms for this problem, and find that it performs well when the batch size is sufficiently large.

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 $(x, y) \in E$. The *Dynamic Topological Order (DTO)* problem involves updating a topological order after an edge insertion. In the *unit-change* version of DTO, the topological order is updated after each edge insertion; in the *batch* version, the topological order is updated after each *batch* of insertions. The DTO problem has many applications and has been studied in the context of constraint-based pointer analysis [21, 9], compilation [14, 17], incremental evaluation of computational circuits [2], constraint-based local search algorithms [16], deadlock detection [4], machine-learning [26], and multiple sequence alignment [24, 25, 7]. For example, in constraint-based pointer analysis, topologically ordering vertices and detecting cycles in the (dynamically changing) constraint graph can dramatically reduce solving time [21, 18, 9]. Since many DTO algorithms (including that studied here) extend naturally to *dynamic cycle detection* [18, 11, 27], such algorithms are key to efficient pointer analysis. Furthermore, in this problem, edges are typically inserted in batches and, hence, there is much to gain from efficient solutions to the batch DTO problem.

Copyright ©2010, Australian Computer Society, Inc. This paper appeared at the Thirty-Third Australasian Computer Science Conference (ACSC2010), Brisbane, Australia. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 102, B. Mans and M. Reynolds, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

Prior Work (Unit Change Problem). In the unit-change problem, the topological order is updated after every edge insertion. A simple approach is to re-compute the topological order from scratch after an edge insertion using a standard topological sort. Using a depth-first search, this yields an $\Theta(v+e)$ time per insertion, where $v = |V|$, $e = |E|$, since every vertex and edge is visited. We can easily improve upon this by first checking whether the inserted edge does, in fact, invalidate the current ordering. In some cases, it will not and, hence, we can avoid work. This gives an $O(v+e)$ runtime, which reflects the fact that some insertions take $\Theta(v+e)$ time, whilst others take $\Theta(1)$ time.

Numerous works have built upon these basic principles to devise more efficient algorithms. In the majority of cases, efficiency is determined by considering an amortised bound on the time taken to process any sequence of k insertions. For example, the cost of inserting k edges using the simple approach outlined above is $O(k \cdot (v+e))$ as, in the worst-case, the cost per insertion is $O(v+e)$.

One of the first works to improve upon this was the algorithm of Marchetti-Spaccamela *et al.* (henceforth, MNR) [15]. This processes a single edge insertion in $O(v+e)$ time, and any sequence of k insertions in $O(ve)$ time. Their algorithm is based upon the following observation:

Definition 1. Let $D = (V, E)$ be a directed acyclic graph and *ord* a valid topological order. For an edge insertion (x, y) , $AR_{xy} = \{k \in V \mid ord(y) \leq ord(k) \leq ord(x)\}$. We call AR_{xy} the Affected Region.

Marchetti-Spaccamela *et al.* showed that only vertices within the affected region need be repositioned to obtain a valid order. Furthermore, whilst the worst-case cost of processing a single insertion is the same as that of the simple approach outlined above, in practice it is very likely that MNR will do much less work as: firstly, both algorithms only do work when the inserted edge invalidates the ordering; secondly, MNR will only visit vertices and edges in the affected region, where as a standard topological sort will always visit *every* edge and vertex in the graph.

One of the most important works in this area is that of Alpern *et al.* [2]. They identified a lower bound, K_{min} , for the unit-change problem:

Definition 2. 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 $\forall a, b \in V. [a \rightsquigarrow b \wedge ord(b) < ord(a) \Rightarrow a \in K \vee b \in K]$. A cover is minimal, written K_{min} , if it is not larger than any valid cover.

Alpern *et al.* provided an algorithm (henceforth, AHRSZ) whose runtime for a single edge insertion was bounded by the number of edges adjacent to members of K_{min} . However, they did not provide

an amortised bound on the time to process any sequence of k insertions. Zhou and Müller improved the space requirements of AHSZ [29]. Katriel and Bodlaender showed, for a slight variant of AHSZ, an $O(\min\{k^{3/2} \log v, k^{3/2} + v^2 \log v\})$ bound on the time to insert k edges [12]. Liu and Chao obtained a tighter bound of $O(k^{3/2} + kv^{1/2} \log v)$ for the Katriel-Bodlaender algorithm [13]. Kavitha and Mathew further improved this to $O(k^{3/2} + k^{1/2}v \log v)$. More recently, Haeupler *et al.* gave yet another variant on the AHSZ algorithm, and achieved an $O(k^{3/2})$ bound on the time to process any sequence of k insertions [10].

Using a different approach, we developed a simpler algorithm (henceforth, PK) and experimentally showed it to be fastest on sparse random graphs [19, 20]. While this has inferior time complexity, compared with those based on the algorithm of Alpern *et al.*, it does have an important advantage: the AHSZ algorithm (and subsequent improvements) rely on an ordered-list data structure [8, 5] which suffers from high overheads in practice, and also from being rather difficult to implement. Ajwani *et al.* also took another approach and obtained an $O(v^{2.75})$ bound with a different algorithm, thus improving upon the result of Katriel and Bodlaender for dense graphs [1]. Finally, Bender *et al.* very recently presented a new algorithm [6] which represents a radical departure from those before, in that it does not maintain an explicit ordering of vertices (which all previous algorithms do). For this, they obtained an $O(v^2 \log v)$ time to process any sequence of k insertions, which improves upon all those before it, particularly for dense graphs.

Prior Work (Batch Problem). The batch version of the DTO problem is slightly more relaxed than the unit-change version. In this case, it is no longer required that the topological order be updated after *every* edge insertion; instead, edge insertions are packaged into batches, with each batch being processed in one go. Thus, the topological order must be updated after every batch of insertions. While this version of the problem arises in practical problems (see e.g. [18, 22, 23]), there have thus far been no specific solutions for it.

As before, a simple approach is to recompute the topological order from scratch after each batch B of insertions (see Algorithm 1). This yields an $O(v + e)$ bound on the time to process a single batch B and, hence, takes $O(k \cdot (v + e))$ time for a sequence of k insertion batches.

Another approach is to simply reuse one of the solutions to the unit-change problem. That is, to process a batch of b insertions as if it were a sequence of b individual insertions. Since, for each of the previous unit-change algorithms, the worst-case time for a single edge insertion is still $O(v + e)$, we arrive at a bound of $O(\min\{kb \cdot (v + e), ?\})$ for a sequence of k insertion batches, each of which has at most b edges. The ? in this bound is a cap on the total cost, as determined by the amortised bound obtained for the unit-change algorithm in question. For example, for MNR, the bound would be $O(\min\{kb \cdot (v + e), ve\})$, whilst for the algorithm of Bender *et al.* [6] it would be $O(\min\{kb \cdot (v + e), v^2 \log v\})$.

We can obtain an even better bound than this by combining both of these approaches together [11]. The idea is simply to run Algorithm 1 in parallel with one of the unit change algorithms when processing a sequence of insertion batches. Then, we simply see which one finishes first and use the topological order it produces (whilst stopping the other immediately). For example, let us consider using MNR here. Since Algorithm 1 takes at most $O(k \cdot (v + e))$ for a sequence

Algorithm 1 ADD_EDGE(B)

```

1: //  $B$  is a batch of updates
2: if  $\exists(x, y) \in B.[ord(y) < ord(x)]$  then
3:     perform standard topological sort

```

of k insertion batches, and MNR takes at most $O(ve)$ for *any* insertion sequence, we arrive at a combined worst-case bound of $O(\min\{k \cdot (v + e), ve\})$ for the parallel algorithm. This improves upon the worst-case bound of either algorithm in isolation. However, whilst this is certainly better in theory, it is also clear that it is not a particularly practical solution. In particular, there will be much redundant work performed by both algorithms as they are, in fact, operating in a very similar fashion. For example, both will perform depth-first traversals of the graph when, in fact, only one traversal is required. Thus, *one desires an algorithm which properly and efficiently combines Algorithm 1 and a unit-change algorithm whilst achieving the same bound without such redundancy.*

Our Contribution. At last, we can now discuss the contributions of this paper:

1. We present the first algorithm (henceforth, PK₂) which genuinely integrates Algorithm 1 with a unit-change algorithm.
2. We provide a proof of correctness for algorithm PK₂.
3. We present results from an empirical comparison of PK₂ against other unit-change algorithms. The results indicate the PK₂ outperforms the other algorithms when the batch size is sufficiently large.

Algorithm PK₂ does not suffer any of the redundancy inherent in the parallel algorithm discussed above. In particular, it *never traverses an edge or visits a vertex more than once when processing a batch of edge insertions*. It will also process a single edge in worst-case $O(v + e)$ time, and any sequence of k batches in worst-case $O(\min\{ve, k \cdot (v + e)\})$ time. To achieve this, we build upon algorithm MNR, primarily because it is the simplest of the unit-change algorithms. *Nevertheless, the bound we obtain on the time to process k batches is still better than all previous algorithms, except for the parallel algorithm discussed already.* For example, the best unit-change algorithm, due to Bender *et al.* [6], requires $O(v^2 \log v)$ to process a single batch. This is a log factor worse than for our algorithm on dense graphs (i.e. when $e = O(v^2)$), and will be more for sparse graphs (i.e. when $e < O(v^2)$). Finally, we hope that this work will motivate future investigation into the batch DTO problem, which has so far been ignored by others.

2 Algorithm PK₂

Before presenting algorithm PK₂, we will first review the operation of algorithm MNR, upon which PK₂ is based.

2.1 Overview of MNR

The algorithm of Marchetti-Spaccamela *et al.* [15] employs an array of size $|V|$ which maps each vertex to a unique integer from $\{1 \dots |V|\}$. In addition, a second array ord^{-1} of size $|V|$ is used, which is the inverse of ord — it maps each index in the order to the corresponding vertex. For an invalidating edge (x, y) ,

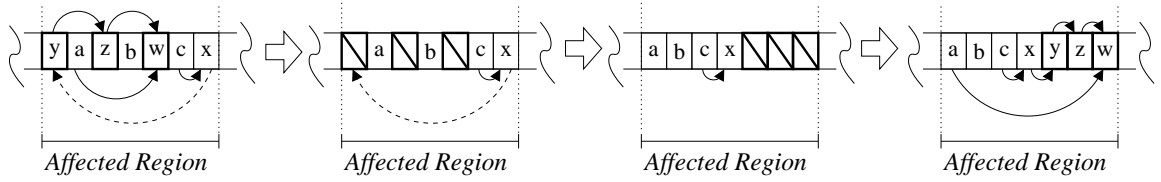


Figure 1: Illustrating the MNR algorithm updating a topological order. The original topological order is on the far left, while that obtained from running MNR is on the far right; in all cases, vertices are laid out in topological order (i.e. increasing in *ord* value) from left to right (hence, the invalidating edge is that drawn with a dashed line). In between, we see the two stages of the MNR algorithm: *discovery* and *shifting*. The former identifies vertices which are out-of-order after the edge insertion, and is implemented using a DFS from y (restricted to the affected region). The latter shifts those vertices found during discovery up the order, so that they now come after x whilst still retaining their original relative orders.

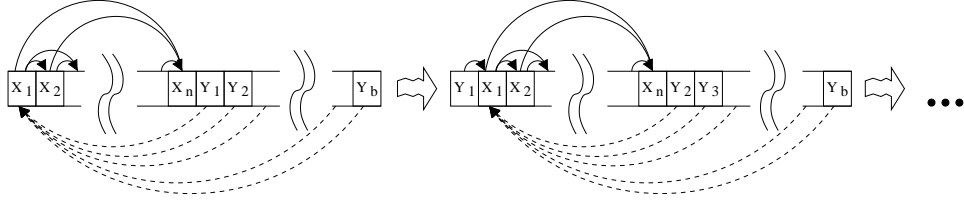


Figure 2: Illustrating a worst-case input for algorithm MNR (left), where the dashed arcs represent the edges to be inserted. The result of processing the first edge insertion (Y_1, X_1) is shown on the right. Here, n is $\Theta(v)$ and there exist $\Theta(e)$ edges of the form (X_i, X_j) , where $i < j$ and $X_1 \rightsquigarrow X_i$. The sequence of new insertions contains b edges of the form (Y_i, X_1) . Assuming these are processed in order, starting from (Y_1, X_1) , then each requires $\Theta(v + e)$ time since every edge reachable from X_1 is traversed by the depth-first search. Thus, MNR needs $\Theta(b(v + e))$ time to solve this graph.

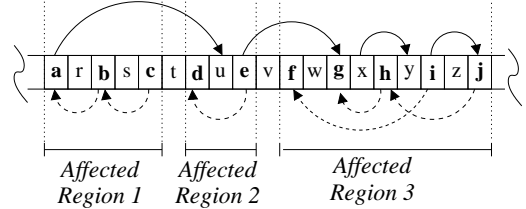
MNR identifies and removes nodes reachable from y in the affected region using a depth-first search (known as *discovery*). Then, it traverses the affected region from the bottom, shifting vacant spaces to the top. Nodes previously removed are now placed in their original order back into the vacant slots. Figure 1 illustrates this. MNR requires $O(v + e)$ time to process an edge insertion. The worst-case occurs when the affected region includes $\Theta(v)$ nodes and $\Theta(e)$ edges reachable from y . Figure 2 illustrates the first edge, (Y_1, X_1) , being processed in a worst-case sequence for MNR. In fact, Marchetti-Spaccamela *et al.* obtained an $O(v e)$ bound on the total time to process any sequence of insertions for MNR [15]. This caps the total cost of processing a batch; hence, even if b was $\Theta(v^2)$ above (e.g. by allowing those of the form (Y_i, X_j)), the runtime would not be $\Theta(v^2 e)$ as might be expected. The proof of this relies on a simple observation that, if an edge $v \rightarrow w$ is traversed as a result of an invalidating edge $x \rightarrow y$, then it won't be traversed again for any other invalidating edge whose tail is x . This is because, having processed $x \rightarrow y$, we have $\text{ord}(x) \leq \text{ord}(v) < \text{ord}(w)$ and, for any subsequent invalidating edge $x \rightarrow z$, we have $\text{ord}(z) \leq \text{ord}(x)$ (otherwise, it isn't invalidating).

2.2 Overview of PK₂

We now present our new algorithm, referred to as PK₂ (since we refer to our earlier algorithm as PK [20]), for the batch DTO problem. The algorithm essentially extends MNR to the batch problem and, when the batch size is 1, they operate in an identical fashion. As with MNR, algorithm PK₂ employs two arrays, ord and ord^{-1} , to map nodes to indices and vice-versa.

The key feature of algorithm PK₂ is that it *never visits or shifts a node more than once* when inserting a batch of edges (unlike MNR). To achieve this, we must alter our notion of the affected region so that overlapping regions are treated as one — so, although

a batch of insertions can still define several affected regions, they are all disjoint and can be processed independently. The following aims to clarify this:



Here, each affected region can be correctly ordered independently of the others, by rearranging its contents. Thus, we extend the definition of an affected region to a batch of overlapping edges by combining their affected regions as follows:

Definition 3. Let $D = (V, E)$ be a DAG and ord a valid topological order. For a set B of overlapping, invalidating edge insertions, the *Affected Region* is denoted AR_B and defined as $\{k \in V \mid b \leq \text{ord}(k) \leq t\}$, where b (resp. t) is the lowest (resp. highest) index of $\{x \mid (x, y) \in B \vee (y, x) \in B\}$.

2.3 Shift Procedure

The first difficulty lies in rearranging an individual affected region without visiting or shifting any node twice. To achieve this goal, we introduce the notion of a *shift set* as follows:

Definition 4. A *frontier pair*, (x, d) , is a pair of nodes in AR_B where $d \rightsquigarrow x$, $\text{ord}(x) < \text{ord}(d)$ and where $\neg \exists z \in AR_B. [z \rightsquigarrow x \wedge \text{ord}(d) < \text{ord}(z)]$. We refer to d as the destination of x .

Informally, a frontier pair (x, d) identifies a vertex x to be reordered and its destination d , which is the vertex furthest up the ordering where $d \rightsquigarrow x$. Our algorithm must ensure x is located above d in the final ordering.

Algorithm 2 SHIFT(i, Q) // i is leftmost position in affected region, Q is a shift queue

```

1:  $n = 0$       // number of nodes temporarily removed from order so far
2: while  $Q \neq \emptyset$  do
3:    $w = \text{ord}^{-1}(i)$       //  $w$  is node at topological index  $i$ 
4:   if  $\text{vacant}(w)$  then
5:      $n = n + 1$  ;  $\text{vacant}(w) = \text{false}$ ;      // reset vacant flag as slot will be occupied by end
6:   else
7:      $\text{allocate}(w, i - n)$ 
8:     // now insert all nodes associated with index  $i$ 
9:      $(v, d) = \text{head}(Q)$ 
10:    while  $Q \neq \emptyset \wedge w = d$  do
11:       $n = n - 1$  ;  $\text{allocate}(v, i - n)$  ;  $\text{pop}(Q)$  ;  $(v, d) = \text{head}(Q)$ 

```

procedure $\text{allocate}(v, i)$

```

12:  $\text{ord}(v) = i$  ;  $\text{ord}^{-1}(i) = v$       // place  $v$  at index  $i$ 

```

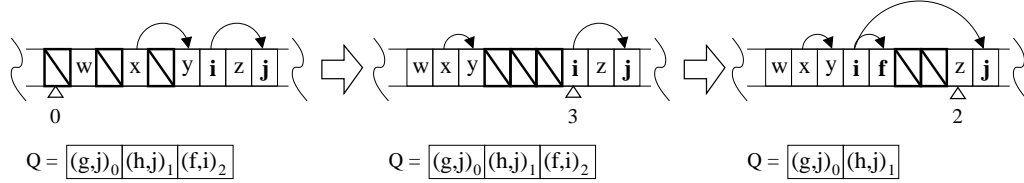


Figure 3: Illustrating the shift procedure of Algorithm 2 operating on affected region 3 from our running example. The topological order after discovery is shown on the left, where the spaces are slots vacated by nodes now on the shift queue Q . In the middle diagram, the algorithm has begun shifting vertices into the correct position; it has reached the destination of the head element on Q . In the final diagram, we see that f has been placed after its destination i , and the algorithm will proceed to push the vacant spaces up the order

Definition 5. For a set B of overlapping, invalidating edge insertions, the shift set is defined as the set of all frontier pairs (x, d) in AR_B .

For example, the shift set for Affected Region 3 in the example above (as $j \rightsquigarrow i$) is: $\{(f, i), (g, j), (h, j)\}$. We can obtain a valid ordering from this by shifting each node immediately right of its destination, whilst topologically sorting those with the same destination (Lemma 1). To formalise this process, we refine our notion of a shift set into that of a *shift queue*:

Definition 6. A shift queue, Q , is a shift set whose members are also totally ordered. More specifically, $\forall (x_1, d_1)_i, (x_2, d_2)_j \in Q. [(ord(d_1) > ord(d_2) \vee (d_1 = d_2 \wedge x_1 \rightsquigarrow x_2)) \Rightarrow i < j]$. The head of the queue (i.e. the pair to be removed first) is that with the highest index.

The shift process operates by scanning the affected region from bottom-to-top whilst shifting vacant slots up the order. During this process, if the current vertex being examined is d and (v, d) is the head of the shift queue, then v is placed into the vacant slot immediately after d ; and, if the next element on the shift queue has the same destination, it is placed immediately after that and so on, until all those with the same destination are placed.

The shift procedure is detailed in Algorithm 2. Figure 3 illustrates the algorithm operating on Affected Region 3 from our example before, assuming the shift queue has been constructed already. Note, for clarity, the value of i on Line 2 is indicated by the triangular marker, whilst the value of n at that point is given below it. In the figure, the left diagram shows the spaces vacated by those now on the shift queue Q . In the middle diagram, the algorithm has begun shifting them up the order and has reached the destination of the head element on Q . In the final diagram, we see that f was placed after its destination i , and that the algorithm is proceeding to shift the remaining vacant spaces up the order.

Lemma 1. Let $D = (V, E)$ be a DAG, ord a valid topological order (with ord^{-1} as its reverse map) and B a set of overlapping, invalidating edge insertions. Given Q , a shift queue for B , Algorithm 2 produces a valid topological order.

Proof. Let ord' be the updated topological order. Suppose ord' is invalid. Then, there is some $x, y \in V$ where $x \rightsquigarrow y$ and $ord'(y) < ord'(x)$. Since Algorithm 2 only repositions nodes within AR_B , it follows that $x, y \in AR_B$ (otherwise, ord was invalid to begin with). There are five cases to consider:

- i) $(y, d_y)_i \notin Q$ and $(x, d_x)_j \notin Q$. In this case, the relative positions of x and y are preserved by Algorithm 2 and, hence, $ord'(y) < ord'(x) \Rightarrow ord(y) < ord(x)$ which is a contradiction.
- ii) $(y, d_y)_i \in Q$ and $(x, d_x)_j \notin Q$. Here, $ord(d_y) < ord(x)$ is needed to get $ord'(y) < ord'(x)$ as y is placed immediately after d_y and the relative positions of d_y and x do not change (note, $(d_y, dd_y) \notin Q$ follows from Definition 5). However, this implies Q is malformed under Definition 5 (as $x \rightsquigarrow y \wedge ord(d_y) < ord(x)$).
- iii) $(y, d_y)_i \notin Q$ and $(x, d_x)_j \in Q$. In this case, $ord(d_x) < ord(y)$ follows from Definition 5 (otherwise, d_x would be y 's destination since $x \rightsquigarrow y$). Again, since x is placed immediately after d_x and the relative positions of d_x and y do not change, we arrive at $ord'(x) < ord'(y)$.
- iv) $(y, d_{xy})_i \in Q$ and $(x, d_{xy})_j \in Q$. Here, $i < j$ follows from Definition 6 and, since vertices are placed in the order popped from Q and that with highest index is popped first, this gives $ord'(x) < ord'(y)$.

```

1:  $Q = \emptyset$  ; sort( $B$ ) // sort invalidating edges into descending order by index of tail
2: for all  $i = 0 \dots |B|$  do
3:    $(x, y) = B[i]$ 
4:   if  $\neg \text{vacant}(y)$  then dfs( $y, \text{ord}(x)$ )
5: return  $Q$ 

```

procedure dfs(v, ub)

```

6:  $\text{vacant}(v) = \text{true}$  ;  $\text{onStack}(v) = \text{true}$ 
7: for all  $(v, s) \in E$  do
8:   if  $\text{onStack}(s)$  then abort // cycle detected
9:   if  $\neg \text{vacant}(s) \wedge \text{ord}(s) < ub$  then dfs( $s, ub$ ) // visit if not already and in  $AR_B$ 
10:  $\text{onStack}(v) = \text{false}$  ; push( $(v, \text{ord}^{-1}(ub)), Q$ )

```

v) $(y, d_y)_i \in Q$ and $(x, d_x)_j \in Q$ and $d_x \neq d_y$. Here, $\text{ord}(d_y) > \text{ord}(d_x)$ follows from Definition 5 (otherwise, d_x would be y 's destination since $x \rightsquigarrow y$) and, hence, $i < j$ from Definition 6. Again, this gives $\text{ord}'(x) < \text{ord}'(y)$, since vertices are placed in the order popped from Q .

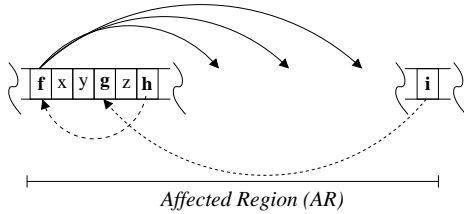
□

Finally, it is fairly easy to see that Algorithm 2 runs in time linear in the size of the affected region.

2.4 Discovery Procedure

The goal of the discovery phase is to construct the shift queue for an affected region without visiting a node or traversing an edge more than once. Recall the discovery procedure of MNR consists of searching from the head of an invalidating edge to identify and mark those which must be shifted past its tail. In the new procedure, we start from the invalidating edge (x, y) with largest $\text{ord}(x)$ value, where x is not already on the shift queue, and search forward from y using a depth-first search; during this, (u, x) is placed onto the shift queue (in post-order) for each vertex u visited. This is repeated until there are no more invalidating edges to process, at which point the shift queue is complete. Observe that choosing the invalidating edge with largest $\text{ord}(x)$ value ensures the correct destination is obtained for each node visited.

Pseudo-code for the discovery procedure is given in Algorithm 3. A subtle aspect of the procedure is the way in which the forward search is pruned. For MNR, each search was restricted to any node within the affected region. For 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, h is the destination of those discovered from f and, hence, those right of h can be ignored. Therefore, Algorithm 3 restricts the search to just those nodes whose index is lower than the current destination (h in this case).

Lemma 2. Assume $D = (V, E)$ is a DAG and ord an array mapping each vertex to a unique index from $\{1 \dots |V|\}$, with ord^{-1} as its reverse map. If a batch B of overlapping, invalidating edge insertions does not introduce a cycle, then Algorithm 3 produces a valid shift queue Q .

Proof. First, we show Q is a valid shift set (i.e. $\forall x, d_x. [(x, d_x) \in Q \iff (x, d_x \in AR_B \wedge d_x \rightsquigarrow x \wedge \text{ord}(x) < \text{ord}(d_x) \wedge \neg \exists z \in AR_B. [z \rightsquigarrow x \wedge \text{ord}(d_x) < \text{ord}(z)])]$) by considering each direction in turn:

\Rightarrow Since d_x is the tail of an edge in B , $d_x \in AR_B$ holds by definition. Since $d_x = \text{ord}^{-1}(ub)$ on Line 11, $\text{ord}(x) < \text{ord}(d_x)$ follows from the condition on Line 9. Now, $\text{dfs}(v, \text{ord}(d_x))$ only explores vertices reachable from d_x , so $d_x \rightsquigarrow x$. Let (u, w) be the last invalidating edge on the path $d_x \rightsquigarrow x$ (it must cross at least one since $\text{ord}(x) < \text{ord}(d_x)$). Then, $\text{ord}(w) \leq \text{ord}(x)$ by definition and, hence, $x \in AR_B$. Finally, we show $\neg \exists z \in AR_B. [z \rightsquigarrow x \wedge \text{ord}(d_x) < \text{ord}(z)]$ by contradiction. Suppose such a z existed. By definition, any invalidating edge (z, w) will be seen on Line 3 before any (d_x, u) , but no $\text{dfs}(w, \text{ord}(z))$ call manages to reach x (otherwise, $d_x = z$). Since $z \rightsquigarrow x$, at least one $\text{dfs}(w, \text{ord}(z))$ call failed to reach x because it encountered nodes vacated by an earlier $\text{dfs}(v, \text{ord}(z'))$ call. This is a contradiction, since it implies $\exists z' \in AR_B. [z' \rightsquigarrow x \wedge \text{ord}(z) < \text{ord}(z')]$.

\Leftarrow We show by contradiction that if $d_x \rightsquigarrow x$, $\text{ord}(x) < \text{ord}(d_x)$ and $\neg \exists z \in AR_B. [z \rightsquigarrow x \wedge \text{ord}(d_x) < \text{ord}(z)]$, then $(x, d_x) \in Q$. Suppose then, that $(x, d_x) \notin Q$. By definition, every invalidating edge (d_x, w) is seen on Line 3, but no $\text{dfs}(w, \text{ord}(d_x))$ call manages to reach x . Since $d_x \rightsquigarrow x$, at least one call failed to reach x because it encountered nodes vacated by an earlier $\text{dfs}(v, \text{ord}(d'_x))$ call. This is a contradiction, since it implies $\exists d'_x \in AR_B. [d'_x \rightsquigarrow x \wedge \text{ord}(d_x) < \text{ord}(d'_x)]$.

Finally, we demonstrate Q must be a valid shift queue. Since invalidating edges with tails higher in ord are seen before those lower down, any (u, d_1) will be pushed onto Q before any (v, d_2) when $\text{ord}(d_2) < \text{ord}(d_1)$. Likewise, since tuples are pushed onto Q in post-order, those with the same destination are pushed in reverse topological order. □

Since Algorithm 3 uses a depth-first search, it follows that it requires $O(v + e)$ time to build the shift queue for a batch of edges insertions. Note, if a merge sort is used on Line 1, then the runtime is actually $O(v + e \cdot \log b)$. However, the log factor can be eliminated using a bucket sort.

Finally, if the insertion batch introduces a cycle, Algorithm 3 will detect this and abort. This property is useful since many applications of DTO algorithms benefit from the detection and elimination of cycles [21, 18, 9]. Informally, this follows from the basic properties of depth-first search and the fact that all vertices in the cycle must lie within the affected region:

```

1:  $E = E \cup B$ ;
2: for all  $(x, y) \in B$  do
3:   if  $\text{ord}(x) < \text{ord}(y)$  then  $B = B - \{(x, y)\}$  // remove forward edges from  $B$ 
4: if  $|B| > 0$  then
5:    $\text{sort}(B)$  // sort invalidating edges into descending order by index of tail
6:    $lb = |V|$  // lowerbound of current region
7:    $s = 0$  // start of current region
8:   for all  $i = 0 \dots |B| - 1$  do
9:      $(x, y) = B[i]$ 
10:    if  $\text{ord}(x) < lb \wedge i \neq 0$  then
11:       $Q = \text{DISCOVER}(\{B[s], \dots, B[i-1]\})$ ; SHIFT( $lb, Q$ )
12:       $s = i$  // start of new region
13:       $lb = \min(\text{ord}(y), lb)$ ;
14:    // Process final region
     $Q = \text{DISCOVER}(\{B[s], \dots, B[|B|-1]\})$ ; SHIFT( $lb, Q$ )

```

Lemma 3. Assume $D = (V, E)$ is a DAG and ord an array mapping each vertex to a unique index from $\{1 \dots |V|\}$, with ord^{-1} as its reverse map. If a batch B of overlapping, invalidating edge insertions introduces a cycle, Algorithm 3 will detect this and abort.

Proof. Suppose B introduces a cycle C , and $\text{dfs}(x, i)$ is first call involving a vertex $x \in C$. Algorithm 3 processes invalidating edges with highest tail first, hence $\forall w \in C. [\text{ord}(w) \leq i]$. Thus, $\text{dfs}(x, i)$ will continue visiting vertices of C until it finds one where $\text{onStack} = \text{true}$ (which must exist as C is a cycle) — at which point it aborts. \square

2.5 Putting it All Together

Pseudo-code for the complete algorithm PK_2 is given in Algorithm 4. This identifies distinct affected regions and processes them using Algorithms 2 and 3. As such, the overall runtime for a single edge insertion is $O(v+e)$ which follows from the individual runtimes of Algorithms 2 and 3. For any sequence of k insertions, we obtain the following amortised bound:

Theorem 1. Let $D = (V, E)$ be a DAG and ord a valid topological order (with ord^{-1} as its reverse map). Then, PK_2 requires $O(\min\{k \cdot (v + e), ve\})$ time to process any sequence of edge insertions split into k batches, where e is the number of edges in the final graph.

Proof. Let the insertion sequence be split into batches B_1, \dots, B_k . Suppose an edge (v, w) is traversed whilst processing some batch B_i . This can only occur if there is an $(x, y) \in B_i$, where $y \rightsquigarrow v$. Furthermore, (v, w) is traversed exactly once whilst processing B_i (this follows because nodes are visited according to a depth-first search). Now, after processing B_i is completed, $\text{ord}(y) < \text{ord}(v)$ will hold. It follows that (v, w) will not be traversed again as a result of any insertion (y, z) (for any z) occurring after this point. This is because the affected region for such an edge must extend to the left of y , but v will always be right of y (from now on). Thus, an edge can be traversed at most v times during any sequence of insertions. Furthermore, since no edge can be traversed more than once whilst processing a batch, an edge can be traversed at most k times when processing k batches. \square

Theorem 1 is a straightforward extension to the proof originally given by Marchetti-Spaccamela *et al.* to show algorithm MNR runs in $O(ve)$ time for any sequence of edge insertions [15]. This bound improves upon the $O(\min\{kb \cdot (v + e), ve\})$ bound obtained by MNR for a sequence of k batches containing at most b edges and, in fact, over all other previously known

algorithms for this problem (except for the parallel algorithm discussed in §1). Furthermore, while PK_2 does the same amount of work as Algorithm 1 in the worst case, there are many situations where PK_2 does much less. This is because for an invalidating insertion, Algorithm 1 always visits every node and every edge, whereas PK_2 visits only those within an affected region. This difference is highlighted by the following:

3 Experimental Study

In this section, we experimentally compare the performance of algorithm PK_2 against various algorithms for this problem: STS (recall Algorithm 1), MNR [15], PK [19] and AHRSZ [2]. As with MNR, neither PK nor AHRSZ offers any benefit to processing edges in batches, rather than one at a time. 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 7. For a DAG with v nodes and e edges, define its density to be $\frac{e}{\frac{1}{2}v(v-1)}$. Thus, it is the ratio of the number of actual edges to the maximum possible.

Definition 8. The model $G_{\text{dag}}(v, p)$ is a probability space containing all graphs having a vertex set $V = \{1, 2, \dots, v\}$ 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 [3]. 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 [28]. The complete implementation, including C++ code for all three algorithms and the random graph generator, is available online at <http://www.ecs.vuw.ac.nz/~djp>.

Figure 4 shows the results of our experiments comparing ACPI for PK₂, MNR, STS, PK and AHRSZ across varying batch sizes at densities 0.001 and 0.01. The plots for MNR, PK and AHRSZ are flat since they obtain no advantage from processing edges in batches. We see that PK₂ is always a better choice than either MNR or STS and, in many cases, offers significant gains over them. When the batch size is one, little difference is observed between MNR and PK₂ which reflects their close relationship. At density 0.01, the gap between these two algorithms has reduced. This is because, 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 [20] for more on this). Thus, there is less chance the affected regions for two invalidating edges will overlap (as there are simply fewer affected regions) which is needed for PK₂ to obtain an advantage over MNR. While PK₂ is the clear winner at density 0.01, compared with PK and AHRSZ, it is less conclusive on the sparser graphs. Certainly, on small batches, it does not perform well by comparison. This is not surprising, since MNR performs poorly on sparse graphs as well and, on small batches, PK₂ and MNR will have similar behaviour. Indeed, given the gains obtained by PK₂ over MNR (on which it is based), it seems quite clear that extending either PK or AHRSZ to deal with batch insertions more efficiently would be valuable. The data also indicates that, for large batches, the performance of STS approaches that of PK₂. 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 PK₂ anyway. Of course, PK₂ can still obtain an advantage because it does not always need to traverse every edge (as STS does).

Figure 5 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 4 and give a strong indication that the results of Figure 4 are not dependent on the experimental machine, rather it is a direct function of the underlying algorithm.

Limitations. The results indicate that PK₂ is always an improvement upon MNR, and that it provides a useful improvement over the other algorithms in certain situations. However, we have not been

able to compare PK₂ against all prior unit-change algorithms (primarily because of the difficulty in implementing these algorithms). However, most of the other unit-change algorithms (i.e. [29, 12, 13, 10]) are minor variants on algorithm AHRSZ, and we would expect them to have very similar performance in practice. The remaining algorithms are that of Ajwani *et al.* [1] and Bender *et al.* [6]. In their paper, Ajwani *et al.* experimentally compare their algorithm against PK, MNR and AHRSZ and find that it only offers an improvement on a particular (artificially constructed) hard class of input graphs. Thus, we would not expect to see their algorithm performing better than PK in any of our experiments. Finally, then, remains the algorithm of Bender *et al.* which is very recent, and does yield the best amortised bound for any sequence of k edge insertions. Bender *et al.* do not report any experimental results for their algorithm, and it remains unclear how efficient it will be in practice. Nevertheless, it would be nice if this algorithm could be empirically evaluated in the future.

4 Conclusion

We have presented the first DTO algorithm which requires $O(\min\{k \cdot (v+e), ve\})$ time to process any sequence of k edge insertion batches. We have experimentally evaluated it against various previous algorithms for this problem, demonstrating that: first, it always outperforms a standard topological sort and the related MNR algorithm; secondly, that it is generally better than the others when the batch size is large enough.

Acknowledgements. Thanks go to Irit Katriel, Gary Haggard and some anonymous referees for helpful comments on earlier drafts of this paper.

References

- [1] D. Ajwani, T. Friedrich, and U. Meyer. An $\tilde{O}(n^{2.75})$ algorithm for online topological ordering. In *Proc. Scandinavian Workshop on Algorithm Theory*, volume 4059 of *LNCS*, pages 53–63. Springer-Verlag, 2006.
- [2] B. Alpern, R. Hoover, B. K. Rosen, P. F. Sweeney, and F. K. Zadeck. Incremental evaluation of computational circuits. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 32–42. ACM Press, 1990.
- [3] A. Barak and P. Erdős. On the maximal number of strongly independent vertices in a random acyclic directed graph. 5(4):508–514, 1984.
- [4] F. Belik. An efficient deadlock avoidance technique. *IEEE TC: IEEE Transactions on Computers*, 39, 1990.
- [5] M. A. Bender, R. Cole, E. D. Demaine, M. Farach-Colton, and J. Zito. Two simplified algorithms for maintaining order in a list. In *Proceedings of the European Symposium on Algorithms (ESA)*, volume 2461 of *Lecture Notes in Computer Science*, pages 152–164. Springer-Verlag, Sept. 2002.
- [6] M. A. Bender, J. T. Fineman, and S. Gilbert. A new approach to incremental topological ordering. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1108–1115. SIAM, 2009.

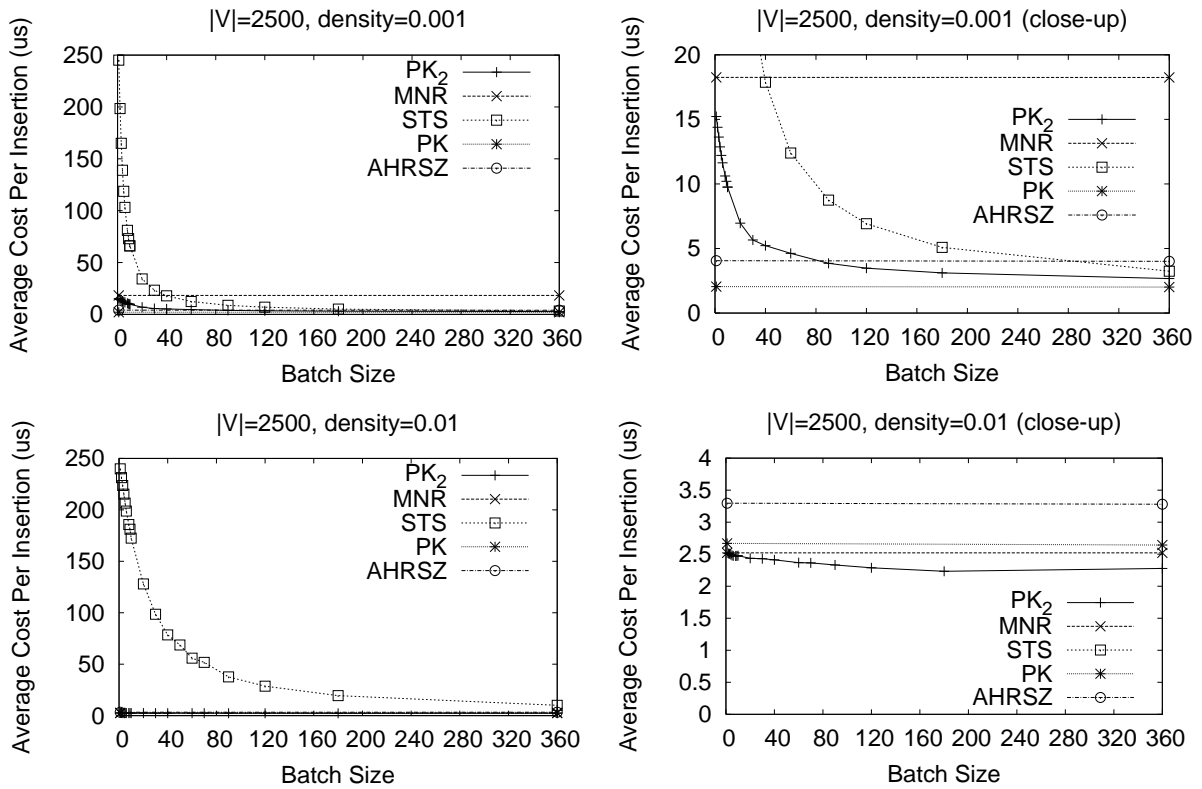


Figure 4: Experimental results looking at the effect of increasing batch size for all three algorithms on random DAGs with 2500 nodes at densities 0.001 and 0.01. For each, batch size is plotted against ACPI and we provide close ups at each density to capture interesting features.

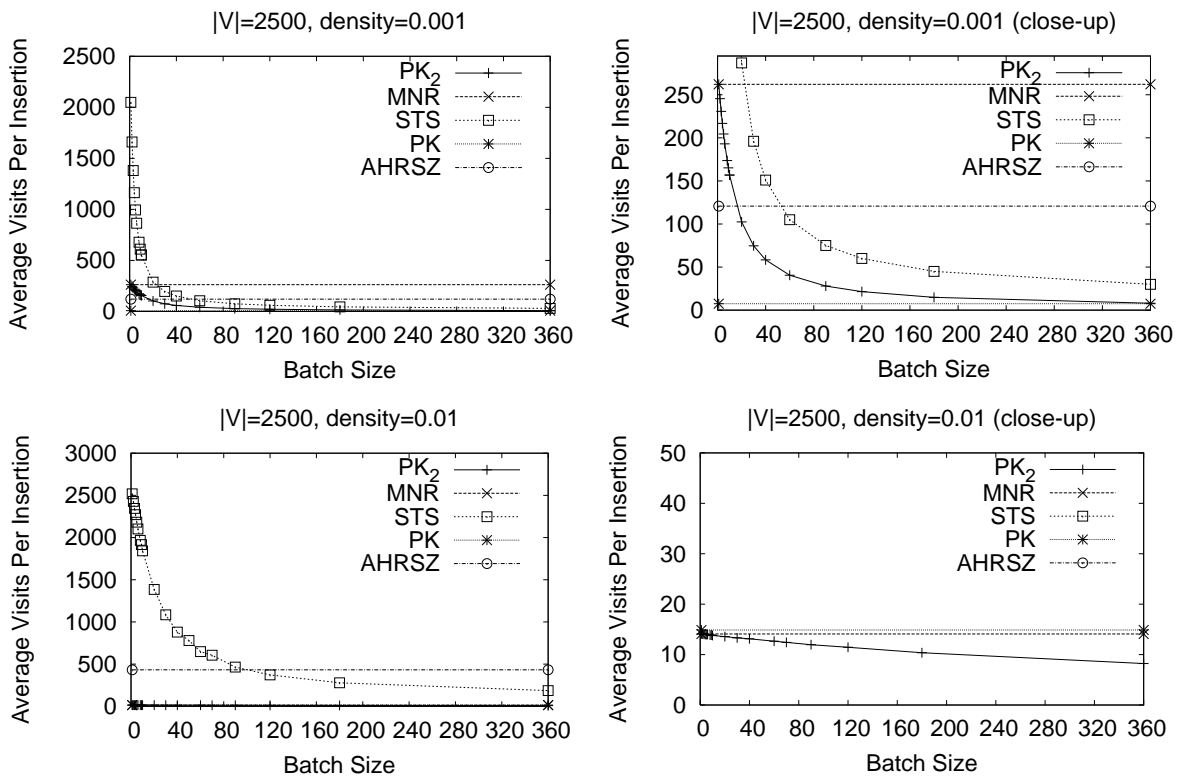


Figure 5: Experimental results looking at the effect of increasing batch size for all three algorithms on random DAGs with 2500 nodes at densities 0.001 and 0.01. For each, batch size is plotted against the average number of nodes and edges visited or shifted when processing an edge insertion. We provide close ups at each density to capture interesting features.

- [7] R. K. Bradley, L. Pachter, and I. Holmes. Specific alignment of structured RNA: stochastic grammars and sequence annealing. *Bioinformatics*, 24(23):2677–2683, 2008.
- [8] P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 365–372. ACM Press, May 1987.
- [9] M. Fährndrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proc. conference on Programming Language Design and Implementation*, pages 85–96. ACM Press, 1998.
- [10] B. Haeupler, T. Kavitha, R. Mathew, S. Sen, and R. E. Tarjan. Faster algorithms for incremental topological ordering. In *International Colloquium on Automata, Languages and Programming (ICALP)*, volume 5125 of *Lecture Notes in Computer Science*, pages 421–433. Springer, 2008.
- [11] B. Haeupler, T. Kavitha, R. Mathew, S. Sen, and R. E. Tarjan. Incremental cycle detection, topological ordering, and strong component maintenance. Technical report, Department of Computer Science, Princeton University, 2008.
- [12] I. Katriel and H. L. Bodlaender. Online topological ordering. In *Proc. ACM Symposium on Discrete Algorithms*, pages 443–450. ACM Press, 2005.
- [13] H.-F. Liu and K.-M. Chao. An $\tilde{O}(n^{2.5})$ -time algorithm for online topological ordering. *Theoretical Computer Science*, 389(1-2):182–189, 2007.
- [14] A. Marchetti-Spaccamela, U. Nanni, and H. Rohnert. On-line graph algorithms for incremental compilation. In *Workshop on Graph-Theoretic Concepts in Computer Science*, pages 70–86, 1993.
- [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] L. Michel and P. V. Hentenryck. A constraint-based architecture for local search. In *Proc. Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 83–100. ACM Press, 2002.
- [17] S. Omohundro, C.-C. Lim, and J. Bilmes. The sather language compiler/debugger implementation. Technical report, International Computer Science Institute, Berkeley, 1992.
- [18] D. J. Pearce. *Some directed graph algorithms and their application to pointer analysis*. PhD thesis, Imperial College, London, UK, 2005.
- [19] D. J. Pearce and P. H. J. Kelly. A dynamic algorithm for topologically sorting directed acyclic graphs. In *Proc. Workshop on Efficient and Experimental Algorithms*, volume 3059 of *LNCS*, pages 383–398. Springer-Verlag, 2004.
- [20] D. J. Pearce and P. H. J. Kelly. A dynamic topological sort algorithm for directed acyclic graphs. *ACM Journal of Experimental Algorithmics*, 11:1.7, 2007.
- [21] D. J. Pearce, P. H. J. Kelly, and C. Hankin. Online cycle detection and difference propagation: Applications to pointer analysis. *Software Quality Journal*, 12(4):309–335, 2004.
- [22] D. J. Pearce, P. H. J. Kelly, and C. Hankin. Efficient field-sensitive pointer analysis for C. *ACM Trans. Prog. Lang. Syst.*, 30(1), 2008.
- [23] F. M. Q. Pereira and D. Berlin. Wave propagation and deep propagation for pointer analysis. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 126–135. IEEE Computer Society, 2009.
- [24] A. S. Schwartz. *Posterior Decoding Methods for Optimization and Accuracy Control of Multiple Alignments*. PhD thesis, University of California, Berkeley, 2007.
- [25] A. S. Schwartz and L. Pachter. Multiple alignment by sequence annealing. *Bioinformatics*, 23(2):24–29, 2007.
- [26] J. Shen, L. Li, and W.-K. Wong. Markov blanket feature selection for support vector machines. In D. Fox and C. P. Gomes, editors, *AAAI*, pages 696–701. AAAI Press, 2008.
- [27] O. Shmueli. Dynamic cycle detection. *Information Processing Letters*, 17(4):185–188, Nov. 1983.
- [28] J. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.
- [29] 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.