Online algorithms for maintaining the topological order of a directed acyclic graph

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 the issue of maintaining the topological order for a directed graph in the presence of edge insertions and deletions. We present a new algorithm and provide empirical data on random graphs comparing it with two existing solutions. In addition, we obtain a marginally improved bounded complexity result over the previously known $O(||\delta||log||\delta||)$. The results show our algorithm to perform better than the rest in all situations except very dense graphs. This we attribute to its simplicity and good theoretical complexity. Our motivation for this work arises from efforts to build efficient pointer analyses.

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. [1]). 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. A partially dynamic algorithm is termed *incremental*/*decremental* if it only supports edge insertions/deletions. The contributions of this paper are as follows:

1. A new fully dynamic algorithm for maintaining the topological order of a directed acyclic graph.

2. An experimental evaluation of this against two previously known algorithms.

1.1 Motivation

The motivation behind this work arises from efforts to speed up pointer analyses. The purpose of such an analysis is to statically determine the target set for all pointer variables in a program. Any solution will always be approximate and the aim is to produce the smallest target set possible for each variable. A pointer analysis can be formulated using simple set-constraints, generated from the program source. A small language is used to describe these constraints:

$$p \supseteq q \mid p \supseteq \{q\} \mid p \supseteq *q \mid *p \supseteq q$$

where p and q are variables and '$*$' is the usual dereference operator. Those involving a dereference are referred to as complex.

To solve a set of these constraints, we formulate them into a *constraint graph* with vertices and edges representing variables and constraints respectively. This idea was first used by Heintze and Tardieu [2]. Initially, complex constraints cannot be fully represented as the solution for a dereferenced variable is at least partially unknown. Instead, edges resulting from them are added during solving and, thus, give rise to a dynamic setting. It turns out that topologically ordering the constraint graph is useful for efficient solving and, hence, an online algorithm is sought after. The reader is referred to [3] for more on this.

1.2 Notation

To simplify the remainder it is useful to clarify our notation here. We assume that $G = (V, E)$ is a directed graph:

Definition 1 *The path relation, $\rightsquigarrow$, holds true if $\forall x, y \in V.[x \rightsquigarrow y \iff x \rightarrow y \in E_T]$, where $G_T = (V, E_T)$ is the transitive closure of G. If $x \rightsquigarrow y$, 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 $E(S) = \{x \rightarrow y \mid x \rightarrow y \in E \wedge (x \in S \vee y \in S)\}$.*

Definition 3 *The extended size of a set of vertices, $K \subseteq V$, is denoted $||K|| = |K| + |E(K)|$. This definition originates from [4].*

1.3 Organisation

The paper will proceed as follows. Section 2 covers related work. Several algorithms for this problem are examined closely in Section 3. Section 4 provides an empirical comparison of these. Finally, we summarise our findings and discuss future work in Section 5.

2 Related Work

The offline topological sorting problem has been widely studied in the past and optimal algorithms with $\Theta(||V||)$ (i.e. $\Theta(|V| + |E|)$) are known (see e.g. [1]).

The problem of maintaining a topological ordering online, however, appears to have received little attention. Indeed, there are only two previously known algorithms which, henceforth, we refer to as AHRSZ [4] and MNR [5]. 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 - For a single edge insertion, it achieves an $O(||\delta||log||\delta||)$ time complexity, where δ is the number of nodes needing reprioritisation [4, 6].

– MNR - Here, an amortised time complexity of $O(|V|)$ over $\Theta(|E|)$ insertions has been shown [5].

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 would 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 arisen [7, 4, 8, 6, 9]. Here, cost is measured in terms of a parameter δ, which captures the change in input and output. In other words, δ measures the amount of work needed to update the solution after some incremental change. For example, an algorithm for the online topological order problem will take as input $< G, ord >$, producing $< G, ord' >$ as output. Thus, the size of the change in input and output (i.e. $|\delta|$) will be the number of vertices whose priority has changed. 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 [6]. 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. [10–15]). Solutions to the latter all employ matrix multiplication in one form or another and this causes problems when dealing with large graphs. For undirected graphs, there has been substantially more work and a survey of this area can be found in [16].

The final area relating to work in this paper is that of random graphs. The standard model of random graphs used in the literature is $G(n, p)$ [17]:

Definition 4 *The model $G(n, p)$ is a probability space containing all graphs having a vertex set $V = \{1, 2, \ldots, n\}$ and edge set $E \subseteq \{V \times V\}$. Each possible edge exists with a probability p independently of any others.*

3 Online Topological Order

We now examine three algorithms for online maintenance of a topological order: AHRSZ, MNR and PK. The latter being our contribution. Before doing this however, we must examine in more detail the complexity parameter δ.

Definition 5 *Let $G = (V, E)$ be a directed 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 6 *Let $G = (V, E)$ be a directed graph and ord a valid topological order. For an edge insertion $x \rightarrow y$, the complexity parameter δ_{xy} is defined as $\{k \in AR_{xy} \mid y \rightsquigarrow k \vee k \rightsquigarrow x\}$.*

In what follows we use δ_{xy} where others have used δ, to aid our presentation. Notice that δ_{xy} will be empty when x and y are already correctly prioritised (i.e. when $ord(x) < ord(y)$). We say that *invalidating* edge insertions are those which cause $|\delta_{xy}| > 0$. To understand how the definition of δ_{xy} originates, we must consider which nodes need to be reprioritised after an edge insertion. The idea of a *minimal cover*, put forward by Alpern *et al.* [4] provides the answer.

Definition 7 *For a directed 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 x and y which are incorrectly prioritised, a cover K must include x or y or both. We say that K is minimal if it is not larger than any valid cover. Although we provide no proof, it is easy enough to see that $K_{xy} \subseteq \delta_{xy}$ for any minimal cover K_{xy} after an insertion $x \rightarrow y$. Our reason then, for choosing δ_{xy} over *minimal cover* as the complexity parameter arises from the simple fact that it allows more useful bounds to be expressed on algorithms MNR and PK. Also, we feel it relates more naturally to the way all three algorithms operate.

3.1 The AHRSZ Algorithm

The algorithm of Alpern *et al.* employs a special data structure, due to Dietz and Sleator [18], to implement the priority space. This structure permits new priorities to be created between existing ones in $O(1)$ worse-case time. We now examine each stage in detail, assuming an invalidating edge insertion $x \rightarrow y$:

Discovery: The set of nodes to be reprioritised is determined by simultaneously searching forward from y and backward from x. Those visited have incorrect priorities and are placed into a set D_A. During this, nodes queued for visitation by the forward (backward) search are said to be on the forward (backward) frontier. At each step the algorithm extends one or both of the frontiers using a procedure which aims at reducing the number of edges explored. The forward (backward) frontier is extended by visiting a member with the lowest (largest) priority. This continues until the two frontiers "meet" — when each node on the forward frontier has a priority greater than any on the backward frontier.

We can think of this algorithm as seeking out those nodes in the affected region which reach x or are reachable from y. The meeting of frontiers complicates the issue and effectively means it can stop before completion. Thus, the worse-case scenario is when the algorithm doesn't stop prematurely and, hence, $D_A = \delta_{xy}$. From this, we arrive at the $O(||\delta_{xy}||log\ ||\delta_{xy}||)$ bound on discovery. The log factor arises from the use of priority queues to implement the frontiers, which we assume are heaps.

Reassignment: The reassignment process also operates in two stages. The first is a depth-first search limited to D_A and computes a ceiling on the new priority for each node, where:

$$ceiling(x) = min(\{ord(y) \mid y \notin D_A \ \wedge\ x \rightarrow y\} \cup \{ceiling(y) \mid y \in D_A \ \wedge\ x \rightarrow y\} \cup \{+\infty\})$$

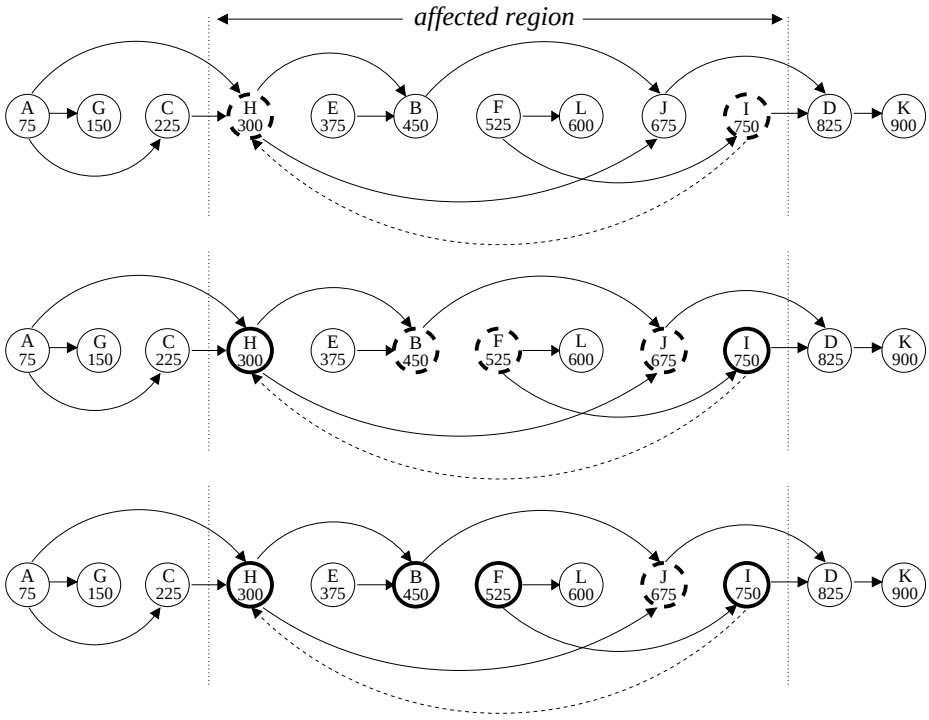

Fig. 1. Illustrating the discovery process of AHRSZ, which uncovers nodes needing reprioritisation. The new edge has a dashed line. Numbers denote priorities. The process consists of a forward search from H and a backward search from I, within the affected area. Nodes with dashed lines are queued for visitation by the forward/backward search and are said to be on the forward/backward frontier. At each step the forward/backward frontier is extended by selecting a node with smallest/greatest priority. Thus, the two frontiers move steadily toward each other and termination occurs when all on the forward frontier have priorities larger than any on the backward frontier. This is true of the third diagram and, hence, node J will not be visited. Nodes with thick borders have been visited and will be reprioritised.

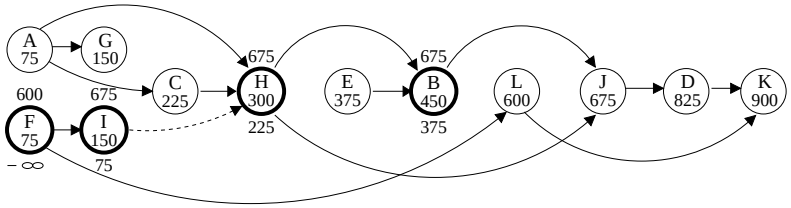

Fig. 2. Showing the graph from Figure 1 after reassignment by AHRSZ. Ceiling and floor values are given above and below each node respectively. The floor on the new priority of a node is the largest (new) priority of any predecessor. Likewise, the ceiling of a node is the minimum of the priorities of unmarked successors and ceilings of marked successors. The floor on F is $-\infty$ because it has no predecessors.

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

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

Once this has been computed the algorithm assigns a new priority, $ord'(k)$, such that $floor(k) < ord'(k) < ceiling(k)$. An $O(|\delta_{xy}|log\ |\delta_{xy}| + |E(\delta_{xy})|)$ bound on the time complexity of 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 D_A are placed onto this queue. Finally, Figures 1 and 2 provide detailed illustrations of the algorithm operating on an example. Note that in Figure 1, we assume that each frontier is extended at each step to simplify the presentation. In reality, a more complex procedure is used which can arise in either or both being extended in a single step.

Complexity: The discovery stage dominates the time complexity, giving an overall bound of $O(||\delta_{xy}||log\ ||\delta_{xy}||)$ for AHRSZ [4, 6].

3.2 The MNR Algorithm

The algorithm of Marchetti-Spaccamela *et al.* differs from the previous by maintaining a total ordering of vertices. This allows the priority space to be implemented using the two arrays $n2i$ and $i2n$ of size $|V|$. The former is the *node-to-index* map and the latter is the converse *index-to-node* map. The priority of a node is therefore its index in the $i2n$ array. Thus, $i2n[0]$ is the vertex with lowest priority and $n2i[0]$ is the priority of vertex zero. Note, we refer to the index of a node in the $i2n$ array as its topological index. Hopefully, it is clear that $n2i$ can be used to implement ord. The purpose of $i2n$, then, is purely to bound the cost of updating $n2i$. The algorithm is presented in Figure 5 and an overview of it operating on our example graph is shown in Figure 3. The two stages are summarised in the following, which assume an invalidating insertion $x \rightarrow y$:

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.

Complexity: We obtain the following (unbounded) complexity result for algorithm MNR: $O(||\delta_{xy}|| + |AR_{xy}|)$.

3.3 The PK Algorithm

We now present our algorithm for maintaining the topological order of a graph online. It achieves the bounded complexity of AHRSZ with the simplicity of MNR. To implement the priority space, we adopt the approach of MNR. For an invalidating insertion $x \rightarrow y$, the algorithm searches forward from y and backward from x to fully identify δ_{xy}. This is similar to the approach of AHRSZ, except that we always perform the full

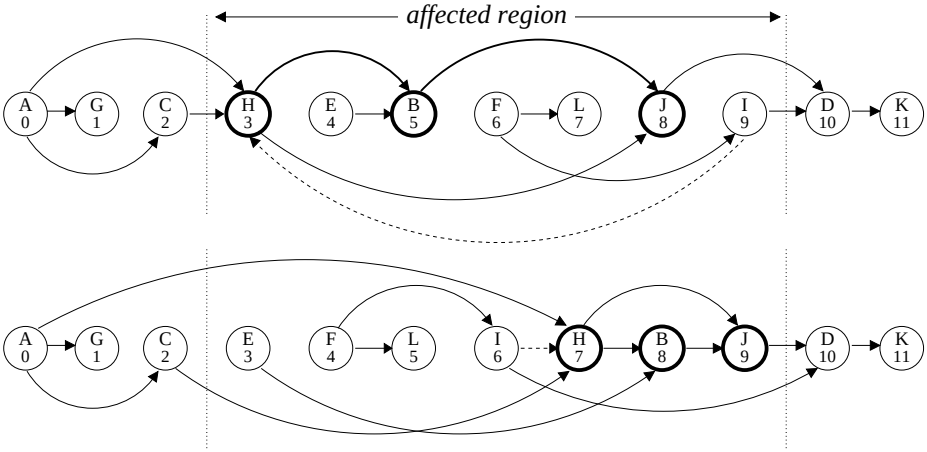

Fig. 3. Showing the two phases of the MNR algorithm updating the same graph as in Figure 1. The new edge is marked with a dashed line. The numbers represent priority values. These correspond directly to indices in the $i2n$ array and, therefore, must always increase contiguously from zero. Note, we have used letters instead of integers to identify vertices purely to simplify the presentation. The top diagram shows those nodes marked during discovery, which is a DFS rooted at H within the affected region. The bottom diagram illustrates the state after reassignment. Notice how each node in the affected area gets a new priority and that all the marked nodes are now located next to each other.

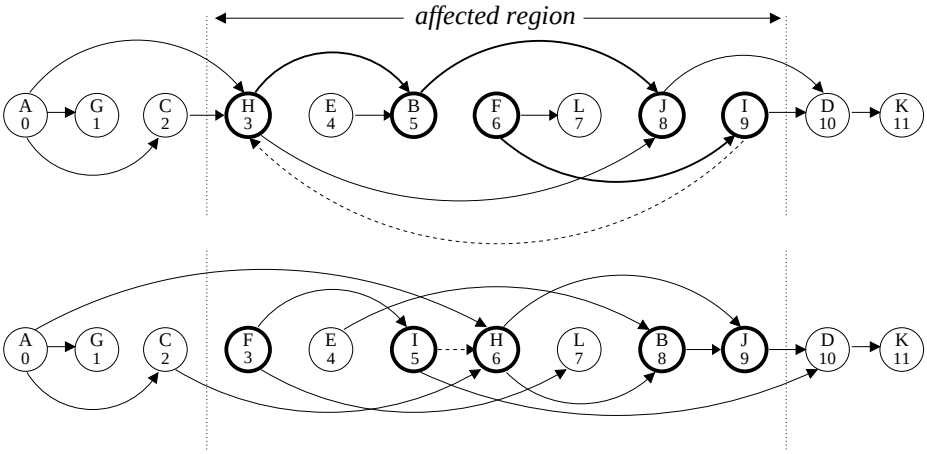

Fig. 4. Showing the two phases of the PK algorithm updating the same graph of Figure 3. Again, the top diagram highlights the nodes marked during discovery with thick borders. These are determined by a forward DFS rooted at H and a backward DFS rooted at I. Notice that more nodes have been marked than were for the other two algorithms. The bottom diagram shows the updated state after reassignment has taken place. Notice that, unlike MNR, unvisited nodes retain their original priority.

```
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
    // visit s if not already and is in affected region
    if s not visited ∧ n2i[s] < ub then dfs(s);

procedure shift()
  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 allocate(w, i − shift);
  for j = 0 to |L| − 1 do allocate(L[j], i − shift); i = i + 1;

procedure allocate(n, index) n2i[n] = index; i2n[index] = n;
```

Fig. 5. The MNR algorithm. This first marks those nodes reachable from y in AR_{xy} and then shifts them to lie immediately after x in $i2n$.

search. We then reprioritise by visiting only the locations of $i2n$ that contain nodes in δ_{xy}. Thus, those in AR_{xy} but not δ_{xy} are never visited, giving a bounded complexity result. Figure 6 presents the algorithm while Figure 4 provides a walk-through illustration. The following points summarise the two stages, again assuming an invalidating insertion $x \rightarrow y$:

Discovery: A forward depth-first search places the topological index of y and those reachable from it into R_F. A backward search places indices of x and those reaching it into R_B. Both are limited to the affected region and, therefore, the total time taken is $\Theta(||\delta_{xy}||)$ since $R_F \cup R_B = \delta_{xy}$.

Reassignment: The two sets are sorted into increasing order, which we assume takes $\Theta(|\delta_{xy}| log\ |\delta_{xy}|)$ time (see e.g. [19]). We then build the new ordering of δ_{xy} using L. Finally, we allocate nodes in δ_{xy} using indices occupied by members of δ_{xy}. This whole procedure takes $\Theta(|\delta_{xy}| log\ |\delta_{xy}|)$ time.

Complexity: PK has a bounded time complexity of $\Theta(|\delta_{xy}| log\ |\delta_{xy}| + |E(\delta_{xy})|)$. To see why this is an improvement upon AHRSZ, we must expand its result according to Definition 3. Doing so gives a bound of $O((|\delta_{xy}| + |E(\delta_{xy})|)\ log\ (|\delta_{xy}| + |E(\delta_{xy})|)$ for AHRSZ. The difference between the two algorithms arises in the discovery phase of AHRSZ, which uses a priority queue to implement each frontier. These will contain nodes visited by their respective search and *any adjacent nodes*. Thus, the two priority queues (frontiers) hold $O(|\delta_{xy}| + |E(\delta_{xy}|))$ nodes between them. In contrast, algorithm PK only sorts $\Theta(|\delta_{xy}|)$ nodes.

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

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

procedure reorder()
  sort(R_B); for i = 0 to |R_B|−1 do w = i2n[R_B[i]]; unmark w; push(w, L);
  sort(R_F); for i = 0 to |R_F|−1 do w = i2n[R_F[i]]; unmark w; push(w, L);
  merge(R_B, R_F, R); for i = 0 to |L|−1 do allocate(L[i], R[i]);
```

Fig. 6. The PK algorithm. The "sort" function sorts an array into increasing order. "merge" combines two arrays into one whilst maintaining sortedness. "allocate" is implemented as for Figure 5. "dfs-b" is similar to "dfs-f" except it traverses in the reverse direction, loads into R_B and compares against lb.

4 Experimental Study

To experimentally compare the three algorithms, we measured their performance over a large number of randomly generated DAGs. We have investigated how insertion cost varies with $|V|$ and $|E|$. Our procedure was to construct a random DAG, with a given number of vertices and outdegree, and measure the time taken to insert 5000 edges. This was repeated 50 times and the average taken to form a data point. To generate a random DAG, we select from the probability space $G_{dag}(n,p)$, a variation on $G(n,p)$:

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 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 by enumerating each edge and inserting with a probability of p.

The data, presented in Figures 7 and 8, was generated on a 900Mhz Athlon based machine with 1Gb of main memory, running Redhat 8.0. The executables were compiled using gcc 3.2, with optimisation level "-O2". Timing was performed using the `gettimeofday` function. The implementation itself was in C++ and took the form of an extension to the *Boost Graph Library* [20]. The source code is available online at `http://www.doc.ic.ac.uk/~djp1/projects/oto-test`. One final point is that our implementation of AHRSZ employs the $O(1)$ amortised (not $O(1)$ worse-case) time structure of Dietz and Sleator [18]. This seems reasonable as they themselves state it likely to be more efficient in practice.

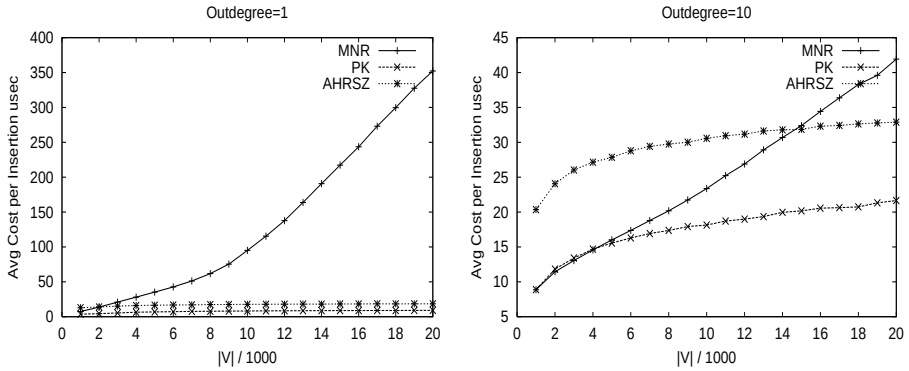

Fig. 7. Experimental data on random graphs with between 1000-60000 vertices. The graphs indicate that MNR has a near-linear complexity with respect to $|V|$, which in turn suggests that the average size of AR_{xy} increases linearly with $|V|$.

4.1 Discussion

The clearest observation from Figures 7 and 8 is that algorithms PK and AHRSZ have similar behaviour, while MNR is quite different. From the examination in Section 3, this was expected as it reflects their complexity bounds. Furthermore, we know that AHRSZ is more complicated than PK and thus, the slight difference between them should be no surprise. And yet, we also know that AHRSZ supports early termination of discovery. Hence, on some inputs it should visit fewer nodes than PK, as demonstrated in Figures 1 and 4. This makes no visible impact and we are unclear as to the reason.

Figure 7: These graphs show the effect of changing $|V|$, while maintaining constant outdegree. For PK and AHRSZ we observe an initial gradient, which quickly tails off. For MNR we see for small $|V|$ it performs well, but in general exhibits linear behaviour. We know from Section 3 that, when $\delta_{xy} \approx AR_{xy}$ and $|E| > |V|$, MNR should do well as it only searches forward and not backward. Remember that the cost of searching increases with $|E|$. Furthermore, it seems reasonable to assume that the average size of AR_{xy} will increase with $|V|$ as, in the worse-case, $|AR_{xy}| = |V|$. In fact, we observed a linear relationship between the two. We conclude that (on average) when $|V|$ is low $\delta_{xy} \approx AR_{xy}$ and, as itincreases, δ_{xy} grows slower than AR_{xy}.

Figure 8: These graphs show the effect of varying outdegree. For PK and AHRSZ we see an initial gradient which eventually levels off, while we note that MNR is worst (best) overall for sparse (dense) graphs. There are several factors which we believe influence the shape of these graphs. Firstly, as outdegree increases we would expect $|AR_{xy}|$ to decrease (on average) as the graph is becoming more ordered. Also, it is reasonable to assume that the *Average Number of Reachable Nodes (ANRN)* from any node will increase with outdegree. This is relevant because it relates to the size of δ_{xy}. We speculate that for small outdegree ANRN is smaller than $|AR_{xy}|$ (on average), but after a certain point is larger. Thus, δ_{xy} should increase until this point, but decrease

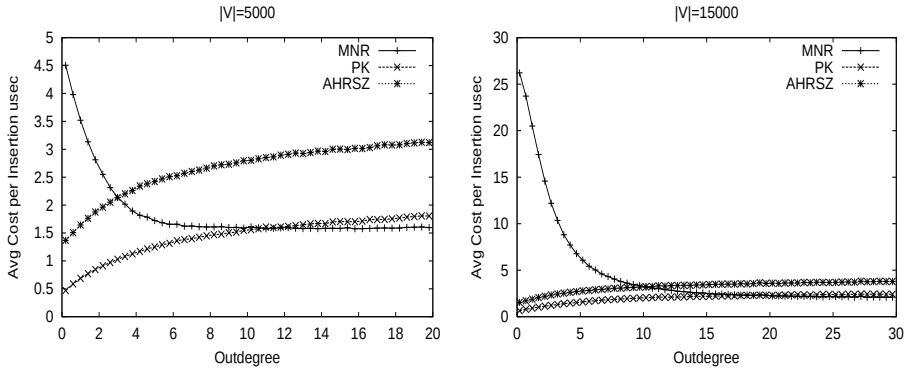

Fig. 8. Experimental data for fixed sized graphs with outdegree between 0.2 and 30. MNR performs well for high outdegree as $AR_{xy} \approx \delta_{xy}$ and it uses the shift algorithm ($O(AR_{xy})$) instead of a second, reverse depth-first search ($O(||\delta_{xy}||)$). In general, all algorithms perform well for high outdegree because the graph is more ordered and, hence, $|AR_{xy}|$ is on average smaller.

afterwards as it is bounded by AR_{xy}. We might expect that this will eventually cause a *negative gradient* for PK and AHRSZ. However, this doesn't appear to happen and we believe it may be counterbalanced by the fact that the cost of discovery increases with $|E|$.

5 Conclusion

We have presented a new algorithm for maintaining the topological order of a graph online and shown it performs better, in general, than those previously known. Furthermore, we have provided the first empirical comparison of algorithms for this problem over a large number of randomly generated acyclic graphs. For the future we are interested in investigating a hybrid of MNR and PK and also using the priority space structure of AHRSZ with the MNR approach. Also, we are aware that the properties of random graphs may not reflect real life structures and, thus, additional data on graphs found in practice would be of benefit.

References

1. J. D. Smith. *Design and Analysis of Algorithms*. PWS-KENT, 1989.
2. N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *Proc. Conference on Programming Language Design and Implementation*, pages 254–263, 2001.
3. 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 (to appear)*, 2003.
4. 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.

5. A. Marchetti-Spaccamela, U. Nanni, and H. Rohnert. Maintaining a topological order under edge insertions. *Information Processing Letters*, 59(1):53–58, 1996.
6. G. Ramalingam and T. Reps. On competitive on-line algorithms for the dynamic priority-ordering problem. *Information Processing Letters*, 51(3):155–161, 1994.
7. T. Reps. Optimal-time incremental semantic analysis for syntax-directed editors. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 169–176, 1982.
8. A. M. Berman. *Lower And Upper Bounds For Incremental Algorithms*. PhD thesis, New Brunswick, New Jersey, 1992.
9. G. Ramalingam. *Bounded incremental computation*, volume 1089 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
10. 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.
11. 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.
12. H. Djidjev, G. E. Pantziou, and C. D. Zaroliagis. Improved algorithms for dynamic shortest paths. *Algorithmica*, 28(4):367–389, 2000.
13. 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.
14. D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic shortest paths and negative cycles detection on digraphs with arbitrary arc weights. In *Proc. European Symposium on Algorithms*, pages 320–331, 1998.
15. 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.
16. G. F. Italiano, D. Eppstein, and Z. Galil. Dynamic graph algorithms. In *Algorithms and Theory of Computation Handbook, CRC Press*. 1999.
17. R. M. Karp. The transitive closure of a random digraph. *RSA: Random Structures & Algorithms*, 1(1):73–94, 1990.
18. 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.
19. D. R. Musser. Introspective sorting and selection algorithms. *Software—Practice and Experience*, 27(8):983–993, August 1997.
20. J. Siek, L. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.