Online algorithms for topological order and strongly connected components

David J. Pearce
Department of Computing,
Imperial College, London,
SW7 2BZ, UK
djp1@doc.ic.ac.uk

Paul H. J. Kelly
Department of Computing,
Imperial College, London,
SW7 2BZ, UK
phjk@doc.ic.ac.uk

ABSTRACT

We consider how to maintain the topological order of a directed acyclic graph (DAG) in the presence of edge insertions and deletions. We present a new algorithm and obtain a marginally improved complexity result over the previously known $O(||\delta||log||\delta||)$. In addition, we provide an empirical comparison against three existing solutions using random DAG's. The results show our algorithm to out perform the others on sparse graphs. Finally, we show how the algorithm can be extended to identify strongly connected components online.

Our motivation for this work arises from efforts to build efficient pointer analyses, where identifying cycles in a dynamic graph has a significant impact on performance.

1. INTRODUCTION

For a directed acyclic graph (DAG), $D = (V, E)$, a topological ordering, ord, maps each vertex to a priority value such that, for all edges $x \rightarrow y \in E$, it is the case that $ord(x) < ord(y)$. For digraphs (i.e. directed graphs), the presence of cycles or *strongly connected components* prohibits any valid topological ordering. To be precise, a *strongly connected component* of a digraph, $G = (V, E)$, is a subgraph $S = (V_s, E_s)$ such that each node in V_s is reachable from every other using only edges in E_s. By collapsing each strongly connected component into a single node we obtain a DAG, often referred to as the *condensation graph*.

There exist well known linear time algorithms for computing the topological order of a DAG and the strongly connected components of a digraph (see e.g. [1]). However, these algorithms are considered *offline* as they compute their 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 supports only 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. A complexity result for this algorithm which improves upon the best previously known result.
3. A new complexity analysis of an existing algorithm, called MNR, by Marchetti-Spaccamela *et al.* [2].
4. An experimental comparison of these two algorithms and another, called AHRSZ, by Alpern *et al* [3].
5. Extensions to both MNR and PK for the incremental strongly connected components problem.

The new algorithm takes it roots from MNR, while exhibiting complexity similar to AHRSZ. Our claim is that it offers an improvement over AHRSZ by a constant factor in both time and space and is easier to implement, due to its simplicity. In particular, it does not need the complicated Deitz and Sleator ordered list structure [4] used by AHRSZ..

1.1 Organisation

The paper will proceed as follows: Section 2, while not essential for the remainder, will examine the motivation behind this work; Section 3 covers related work; in Section 4 we begin by examining the complexity parameters used as a basis for comparing the algorithms; Section 4.1 presents our new algorithm and Sections 4.2 details MNR and provides a new complexity analysis of it. This facilitates a comparison against PK and AHRSZ and also helps explain its observed behaviour; Section 4.3 covers AHRSZ and Section 4.4 identifies where it loses out to PK; Section 5 details our experimental work. This includes a comparison of the three algorithms and a study examining when the standard offline algorithm should be used; finally, we summarise our findings and discuss future work in Section 7.

$$[trans]\quad \frac{\tau_1 \supseteq \{\tau_2\} \quad \tau_3 \supseteq \tau_1}{\tau_3 \supseteq \{\tau_2\}} \qquad [deref_1]\quad \frac{\tau_1 \supseteq *\tau_2 \quad \tau_2 \supseteq \{\tau_3\}}{\tau_1 \supseteq \tau_3}$$

$$[deref_2]\quad \frac{*\tau_1 \supseteq \tau_2 \quad \tau_1 \supseteq \{\tau_3\}}{\tau_3 \supseteq \tau_2} \qquad [deref_3]\quad \frac{*\tau_1 \supseteq \{\tau_2\} \quad \tau_1 \supseteq \{\tau_3\}}{\tau_3 \supseteq \{\tau_2\}}$$

Figure 1: A simple inference system for pointer analysis

2. MOTIVATION

The motivation behind this work arises from efforts to speed up pointer analyses. The purpose of such an analysis is to determine the target set for all pointer variables in a program, without executing it. This information is useful for, among other things, compiler optimisations, automatic parallelisation and error checking tools.

A pointer analysis can be formulated using simple set constraints, generated from the program source. A small language is used for this purpose, where the domain of variables is denoted by VAR. Thus, the constraints take the form:

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

where p and q are variables from VAR and '$*$' is the usual dereference operator. Those involving a dereference are referred to as complex. A solution to a set of these constraints is an assignment to each variable from $\mathcal{P}(VAR)$, such that all constraints are satisfied. Thus, for example, if $p \supseteq \{q\}$ then q is in the solution of p and we say that p *points to* q. Any such solution will always be approximate [5] and the aim is to produce the smallest target set possible for each variable.

The solution is obtained by exhaustively deriving all facts under the simple inference system shown in Figure 1. To help put this into context, consider the following example:

`int a,*p,*q,**x;`			
`p=&a;`	(1)	$p \supseteq \{a\}$	
`y=&q;`	(2)	$y \supseteq \{q\}$	
`x=y;`	(3)	$x \supseteq y$	
`*x=p;`	(4)	$*x \supseteq p$	
	(5)	$x \supseteq \{q\}$	$(trans, 2+3)$
	(6)	$q \supseteq p$	$(deref_2, 4+5)$
	(7)	$q \supseteq \{a\}$	$(trans, 1+6)$

Here, we see some simple C statements (left) and their associated constraints (right). Below the line, we see constraints derived using the inference rules of Figure 1. The problem of exhaustively deriving all facts under this system can be reduced to the problem of dynamic transitive closure [6] and, thus, the best known bound on time complexity is $O(n^3)$.

To solve a constraint set (i.e. derive all facts), we formulate them into a *constraint graph* with vertices and edges representing variables and constraints respectively. Thus, constraint $a \supseteq b$ becomes edge $a \leftarrow b$. This idea was first used by Heintze and Tardieu [6]. Initially, complex constraints cannot be represented as the solution for the dereferenced variable is at least partially unknown. Instead, they result in edges being added to the graph during the solving process (via the $deref$ rules). So, in the above example, the constraint $*x \supseteq p$ is not initially represented by an edge in the constraint graph. However, it will lead to the edge $q \leftarrow p$ being added when (6) is derived.

One interesting observation about this system is that variables involved in a strongly connected component (e.g. $a \supseteq b \supseteq a$) must, by definition, have the same final solution. Exploiting this property by collapsing these cycles into single nodes leads to a significant reduction in computation time. However, the dynamic nature of the constraint graph requires an online solution and, hence, is the reason behind this work. The reader is referred to [7].

3. RELATED WORK

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

Definition 1. The path relation, $\rightsquigarrow$, holds 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 [3].

The offline topological sorting problem has been widely studied and optimal algorithms with $\Theta(||V||)$ (i.e. $\Theta(|V| + |E|)$) are known (see e.g. [1]). Similarly, an $\Theta(||V||)$ algorithm for finding the strongly connected components of a digraph was presented by Tarjan [8]. A few minor improvements to this have since been proposed [9, 10], although the complexity bound remains.

For the issue of maintaining online the strongly connected components of a digraph, we are aware of only one previously known algorithm, due to Fähndrich *et al.* [11]. This operates by exhaustively searching from y, when a new edge $x \rightarrow y$ is added, to determine whether x is reachable and, hence, a cycle has been introduced. This approach is inferior to the algorithms we present later which, by maintaining a topological ordering of nodes, can prune this search dramatically. However, it is interesting to note that their motivation is similar to ours.

The problem of maintaining a topological ordering online also appears to have received little attention. Indeed, there

are only two existing algorithms which, henceforth, we refer to as AHRSZ [3] and MNR [2]. We have implemented both and will detail their working in Section 4. 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 [3, 12].
- MNR - Here, an amortised time complexity of $O(|V|)$ over $\Theta(|E|)$ insertions has been shown [2].

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

In an effort to find a simple way of comparing online algorithms the notion of *bounded complexity analysis* has been proposed [13, 3, 14, 12, 15]. 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 $< D, ord >$, producing $< D, ord' >$ 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 [12]. 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. [16, 17, 18, 19, 20, 21]). 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 [22].

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 $G(n,p)$ [23]:

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.

4. ONLINE TOPOLOGICAL ORDER

We now examine three algorithms for online maintenance of a topological order: PK, MNR and AHRSZ. The first 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.* [3] provides the answer.

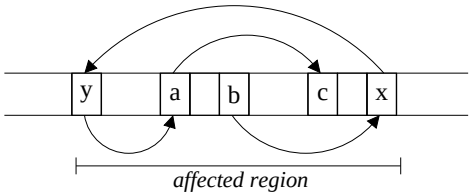
Definition 7. For a directed graph $G = (V, E)$ and an invalidated topological order ord, the set K of vertices is a cover if $\forall x, y \in V.[x \rightsquigarrow y \wedge ord(y) < ord(x) \Rightarrow x \in K \vee y \in K]$.

This states that, for any connected 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.

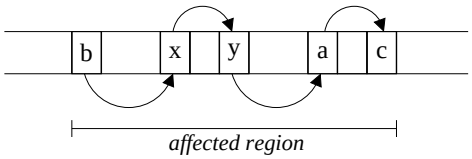
4.1 The PK Algorithm

We now present our algorithm for maintaining the topological order of a graph online. As we will see in the coming Sections, it is similar in design to MNR, but achieves a bounded complexity result which MNR does not. For a DAG D, the algorithm implements the topological ordering, ord, using an array of size $|V|$, called the *node-to-index* map or $n2i$ for short. This maps each vertex to a unique integer, such that for any edge $x \rightarrow y$ in D, $n2i[x] < n2i[y]$. Thus, when an invalidating edge insertion $x \rightarrow y$ is made, the algorithm must update $n2i$ to preserve the topological order property. The key insight here is that we can do this by simply reorganising nodes in δ_{xy}. That is, in the new ordering, $n2i'$, nodes in δ_{xy} are repositioned so as ensure a valid topological ordering, *using only positions previously held by members of* δ_{xy}. All other nodes remain unaffected.

For example, consider the following situation, caused by an invalidating edge $x \rightarrow y$:



Here, nodes are laid out in topological order (i.e. increasing in $n2i$ value from left to right) with members of δ_{xy} shown. As $n2i$ is a total and contiguous ordering, the gaps must contain nodes, which we omit to simplify the discussion. The affected region contains all nodes (including those not shown) between y and x. Now, we can obtain a correct topological ordering by moving y, a, c up the order and x, b down it, giving:

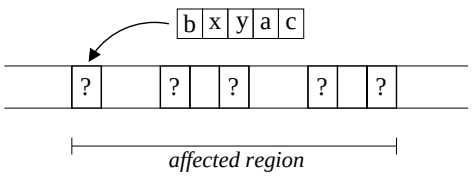


Let us partition the nodes of δ_{xy} into two sets: R_F and R_B. The former contains y and those reachable from it, whilst R_B contains x and those reaching it. We can now make an important observation:

LEMMA 1. *Assume $D = (V, E)$ is a DAG and ord a valid topological order. Let $x \rightarrow y$ be an invalidating edge insertion, which does not introduce a cycle. If $R_F = \{z \in AR_{xy} \mid z = y \vee y \rightsquigarrow z\}$ and $R_B = \{z \in AR_{xy} \mid z = x \vee z \rightsquigarrow x\}$, then no edge exists from any $a \in R_F$ to any $b \in R_B$.*

PROOF. Suppose such an edge, $a \rightarrow b$, existed. This would make x reachable from y as, by definition of R_F and R_B, y reaches a and b reaches x. Hence, the new edge $x \rightarrow y$ causes a cycle, which is a contradiction. □

Using this we can begin to understand how the algorithm works: it first identifies R_B and R_F. Then, it pools the indices occupied by their nodes and, starting with the lowest, allocates increasing indices first to members of R_B and then R_F. The key here is that the relative order of nodes in R_B is preserved and likewise for R_F. So, in the above example, $R_B = \{b, x\}$ and $R_F\{y, a, c\}$ and the algorithm proceeds by allocating b to the lowest available index, like so:



after this, it will allocate x to the next lowest index, then y and so on.

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

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

procedure dfs-b(n)
  mark n as visited;
  R_B ∪= {n};
  forall w→n ∈ E do
    // is w unvisited and in affected region?
    if w not visited ∧ lb<n2i[w] then dfs-b(w);

procedure reorder()
  // sort sets to preserve original order of elements
  sort(R_B); sort(R_F);
  // load R_B onto array L first
  for i = 0 to |R_B|−1 do
    w = R_B[i]; R_B[i] = n2i[w];unmark w; push(w, L);
  // now load R_F onto array L
  for i = 0 to |R_F|−1 do
    w = R_F[i]; R_F[i] = n2i[w];unmark w; push(w, L);
  merge(R_B, R_F, R);
  // allocate nodes in L starting from lowest
  for i = 0 to |L|−1 do n2i[L[i]] = R[i];
```

Figure 2: The PK algorithm. The "sort" function sorts an array such that x comes before y iff $n2i[x] < n2i[y]$. "merge" combines two arrays into one whilst maintaining sortedness. "dfs-b" is similar to "dfs-f" except it traverses in the reverse direction, loads into R_B and compares against lb.

The algorithm is presented in Figure 2 and the following summarises the two stages:

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

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

Algorithm PK has time complexity $\Theta((|\delta_{xy}|log\ |\delta_{xy}|)+||\delta_{xy}||)$. This improves upon the existing bound of $O(||\delta_{xy}||log||\delta_{xy}||)$ obtained for AHRSZ [3]. Section 4.3 will examine the reasons behind this improvement in more detail. Finally, we provide the correctness proof:

LEMMA 2. *Assume $D = (V, E)$ is a DAG and $n2i$ an array, mapping vertices to unique values in $\{0 \ldots |V|-1\}$, which is a valid topoligical order. If an edge insertion $x \to y$ does not introduce a cycle, then algorithm PK obtains a correct topolocial ordering.*

PROOF. Let $n2i'$ be the new ordering found by the algorithm. To show this is a correct topological order we must show, for any two vertices a, b where $a \to b$, that $n2i'[a] < n2i'[b]$ holds. Let $reaches(v) = \{x \mid x = v \vee v \rightsquigarrow x\}$. An important fact to remember is that the algorithm only uses indices of those in δ_{xy} for allocation. Therefore, $z \in \delta_{xy} \Rightarrow n2i[y] \leq n2i'[z] \leq n2i[x]$. There are seven cases to consider:

Case 1: $a \in reaches(x) \wedge n2i[a] > n2i[x]$. Here neither a or b have been moved as they lie outside affected region. Thus, $n2i[a] < n2i[b] \Rightarrow n2i'[a] < n2i'[b]$.

Case 2: $a \in reaches(x) \wedge n2i[a] = n2i[x]$. We know $x = a$ and $n2i'[x] \leq n2i[x] < n2i[b]$, as x had highest index available to any in δ_{xy}. Also, b outside affected region, so $n2i'[b] = n2i[b]$.

Case 3: $a \in reaches(x) \wedge n2i[a] < n2i[x]$. Here, a reachable from x only along $x \to y$. Thus, $a \in reaches(y)$ and so $a \in \delta_{xy}$. If b outside affected region then $n2i'[a] \leq n2i[x] < n2i[b]$ and $n2i[b] = n2i'[b]$. Otherwise, $a, b \in R_F$ and their relative order is preserved in $n2i'$ by sorting.

Case 4: $y \in reaches(b) \wedge n2i[b] < n2i[y]$. Similar to case 1 as a and b outside affected region.

Case 5: $y \in reaches(b) \wedge n2i[b] = n2i[y]$. We know, $y = b$ and $n2i[a] < n2i[y] \leq n2i'[y]$, as y had lowest index available to any in δ_{xy}. Also, a outside affected region, so $n2i'[a] = n2i[a]$.

Case 6: $y \in reaches(b) \wedge n2i[b] > n2i[y]$. Here, b reaches y along $x \to y$. Hence, $x \in reaches(b)$ and so $b \in \delta_{xy}$. If a outside affected region then $n2i[a] < n2i[y] \leq n2i'[b]$ and $n2i'[a] = n2i[a]$. Otherwise, $a, b \in R_B$ and their relative order is preserved in $n2i'$ by sorting.

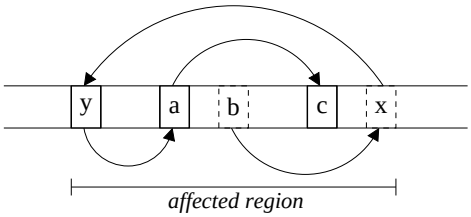
Case 7: $a \notin reaches(x) \wedge y \notin reaches(b)$. By Definition 6 $a, b \notin \delta_{xy}$ and so, they are not repositioned. Thus, $n2i[a] < n2i[b] \Rightarrow n2i'[a] < n2i'[b]$. □

4.2 The MNR Algorithm

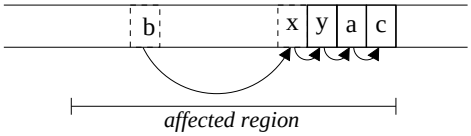
The algorithm of Marchetti-Spaccamela *et al.* operates in a similar way to PK by using a total ordering of vertices. This time two arrays, $n2i$ and $i2n$, of size $|V|$ are used with $n2i$ as before. The second array $i2n$, is the reverse mapping of $n2i$, such that $i2n[n2i[x]] = x$ holds and its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

```
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
      // w reachable from y so will reposition after x
      unmark w; push(w, L); shift=shift+1;
    else allocate(w, i − shift);
  //now place y and nodes reachable from it
  for j = 0 to |L|−1 do
    allocate(L[j], i−shift); i=i+1;

procedure allocate(n, i)
  // place n at index i
  n2i[n] = i; i2n[i] = n;
```

Figure 3: The MNR algorithm. This first marks those nodes reachable from y in AR_{xy} and then shifts them to lie immediately after x in $i2n$.



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



Notice that these nodes always end up alongside x and that, unlike PK, each node in the affected region receives a new position. We can see that this has achieved a similar effect to PK as every node R_B must have lower index than any in R_F.

For completeness, the algorithm is presented in Figure 3 and the two stages are summarised in the following, which assumes an invalidating insertion $x \to y$:

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

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

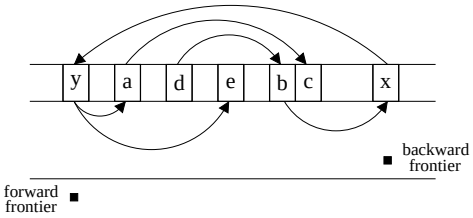
Thus we obtain, for the first time, the following complexity result for algorithm MNR: $O(||\delta_{xy}|| + |AR_{xy}|)$. This highlights an important difference in the expected behaviour between PK and MNR as the affected region (AR_{xy}) can contain many more nodes than δ_{xy}. Thus, we would expect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

4.3 The AHRSZ Algorithm

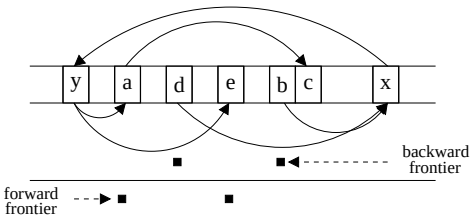
The algorithm of Alpern *et al.* employs a special data structure, due to Dietz and Sleator [4], to implement a priority space which permits new priorities to be created between existing ones in $O(1)$ worse-case time. This highlights the main difference between this and the other two, which use arrays and integers to implement the priority space. The algorithm operates in two stages with the first, like PK, consisting of a forward and backward search. This time, however, it is the minimal cover K_{xy} (recall Definition 7) which is being identified and not δ_{xy} in full.

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. 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 the frontiers toward each other. The forward (backward) frontier is extending by visiting a member with the lowest (largest) priority. The following diagrams aim to clarify this:



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



Here we see that, for example, the forward frontier has been extended by visiting y and this results in a, e being added and y removed. In the next step, a will be visited as it has the lowest priority of any on the frontier. Likewise, the backward frontier will be extended next time by visiting b as it has the *largest* priority. Thus, we see that the two frontiers are moving toward each other and, indeed, the search stops when they "meet" — when each node on the forward frontier has a priority greater than any on the backward frontier. An interesting point here is that the frontiers may meet before R_B and R_F have been fully identified. Thus, the discovery stage may identify fewer nodes than that of algorithm PK. Also, it is important to realise that AHRSZ has no notion of the affected region and, thus, can place nodes outside it onto a frontier. However, such nodes are never visited.

The worse-case scenario is when all members of δ_{xy} are visited. Thus, we get an $O(||\delta_{xy}||log\ ||\delta_{xy}||)$ 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 of those visited during discovery and computes a ceiling on the new priority for each node, where:

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

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

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

Note that, $ord'(x)$ is the topological ordering being generated. Once the floor has been computed the algorithm assigns a new priority, $ord'(k)$, such that $floor(k) < ord'(k) < ceiling(k)$. An $O(|\delta_{xy}|log\ |\delta_{xy}|+|E(\delta_{xy})|)$ 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.

The discovery stage dominates the time complexity, giving an overall bound of $O(||\delta_{xy}||log\ ||\delta_{xy}||)$ for AHRSZ [3, 12].

4.4 How PK wins

We can now see that the difference between the complexity of PK and AHRSZ arises from the use of priority queues in the latter to implement the frontiers. In contrast, algorithm PK sorts the visited nodes once discovery is complete. However, it seems reasonable to conclude that AHRSZ could be modified to achieve the improved bound by, firstly, ensuring nodes outside the affected region are never placed onto a frontier. Secondly, by providing a constant time check for frontier membership.

5. 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 $|V|$, $|E|$ and batch size. The latter relates to the processing of mutiple edges together. Although none of

the algorithms we discuss offer any advantage from processing multiple edges at once, the standard offline topological sort does, and it is interesting to consider when it becomes economical to use. 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. Note, non-invalidating edges were included in our measurements. 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 probability p.

The data, presented in Figures 4, 5 and 6, 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* [24]. The source code is available online at `http://www.doc.ic.ac.uk/~djp1/projects/oto-test`. Our implementation of AHRSZ employs the $O(1)$ amortised (not $O(1)$ worse-case) time structure of Dietz and Sleator [4]. This seems reasonable as they themselves state it likely to be more efficient in practice.

5.1 Discussion

The clearest observation from Figures 4 and 5 is that algorithms PK and AHRSZ have similar behaviour, while MNR is quite different. From the examination in Section 4, 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.

Figure 4: These graphs show the effect of changing $|V|$, while maintaining constant outdegree. Looking at the leftmost graphs, we observe an initial gradient for PK and AHRSZ, which quickly tails off. For MNR we see at small $|V|$ it performs well, but in general exhibits linear behaviour. The rightmost graphs measure average size of δ_{xy} and AR_{xy}. They indicate that AR_{xy} has linear complexity in $|V|$, while δ_{xy} does not, and show a strong resemblence with the runtime behaviour of the three algorithms. The curve for δ_{xy} is perhaps the most interesting feature of these graphs, although we cannot explain it.

Figure 5: 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. The two graphs on the right go someway toward explaining this behaviour. They show that, at some point, $||\delta_{xy}||$ begins to dominates over $|AR_{xy}|$. Thus, the behaviour of MNR is governed initially by $|AR_{xy}|$ and then by $|\delta_{xy}|$. However, it achieves a speedup over PK because it only performs one depth-first search instead of two.

```
procedure dfs(n) // new edge is x → y
  mark n as visited;
  forall n → s ∈ E do
    // visit s if not already and is in affected region
    if s not visited ∧ n2i[s] < ub then dfs(s);
    // back propagate in_comp information
    in_component(n) = in_component(n) ∨ in_component(s);
```

Figure 7: Illustrating how the depth-first search from Figure 3 can be extended to back propagate in_component information.

There are two other interesting features of this data. Firstly, we see that $|AR_{xy}|$ goes down as outdegree increases. We suspect that this can be explained by considering that, as $|E|$ increases, the graphs are becoming more ordered. Thus, the number of distinct chains is decreasing and, hence, the number of invalidating edges within the same chain is increasing. The other interesting aspect of the data is that we observe both a positive and negative gradient for $|\delta_{xy}|$. This is expected as the average number of nodes reachable from any will increase with $|E|$. Thus, we would expect $|\delta_{xy}|$ to increase accordingly. However, $|\delta_{xy}|$ is also governed by the size of the affected region. Thus, as $|AR_{xy}|$ has a negative gradient we must eventually expect δ_{xy} to do so as well. Certainly, when $|AR_{xy}| \approx |\delta_{xy}|$, this must be the case. In fact, the data indicates that the downturn happens someway before this. However, although $|\delta_{xy}|$ decreases, the increasing number of edges appears to counterbalance this, as we observe that $||\delta_{xy}||$ does not exhibit a negative gradient.

Figure 6: These graphs compare an offline topological sort (implemented using the depth-first search approach) to those we are studying. They show a significant advantage is to be gained from using the online algorithms when the batch size is small. Indeed, the data suggests that the online algorithms compare favourably even for large batch sizes. It is important to realise here that the online algorithms can only process one edge at a time. Thus, their graphs are flat as they cannot obtain an advantage from processing edges in batches.

6. ONLINE STRONG COMPONENTS

In this section we show how algorithms PK and MNR can be extended to the problem of incrementally detecting strongly connected components. In both cases, the approach is very similar and based upon a simple observation: *if a new edge $x \to y$ introduces a cycle then x must be visited during a forward depth-first search from y.*

Thus, it is easy enough to tell whether a cycle has been created during the discovery stage of each algorithm. The question is, how can we identify the members of that cycle? By definition, any node z on a path from y to x must be in the cycle and any other node is not. Therefore, we maintain an extra bit of storage for each node, referred to as *in_component*. Initially, this is false for all nodes and, before starting the forward search from y, we set $in\_component(x) = true$. The idea now is to back-

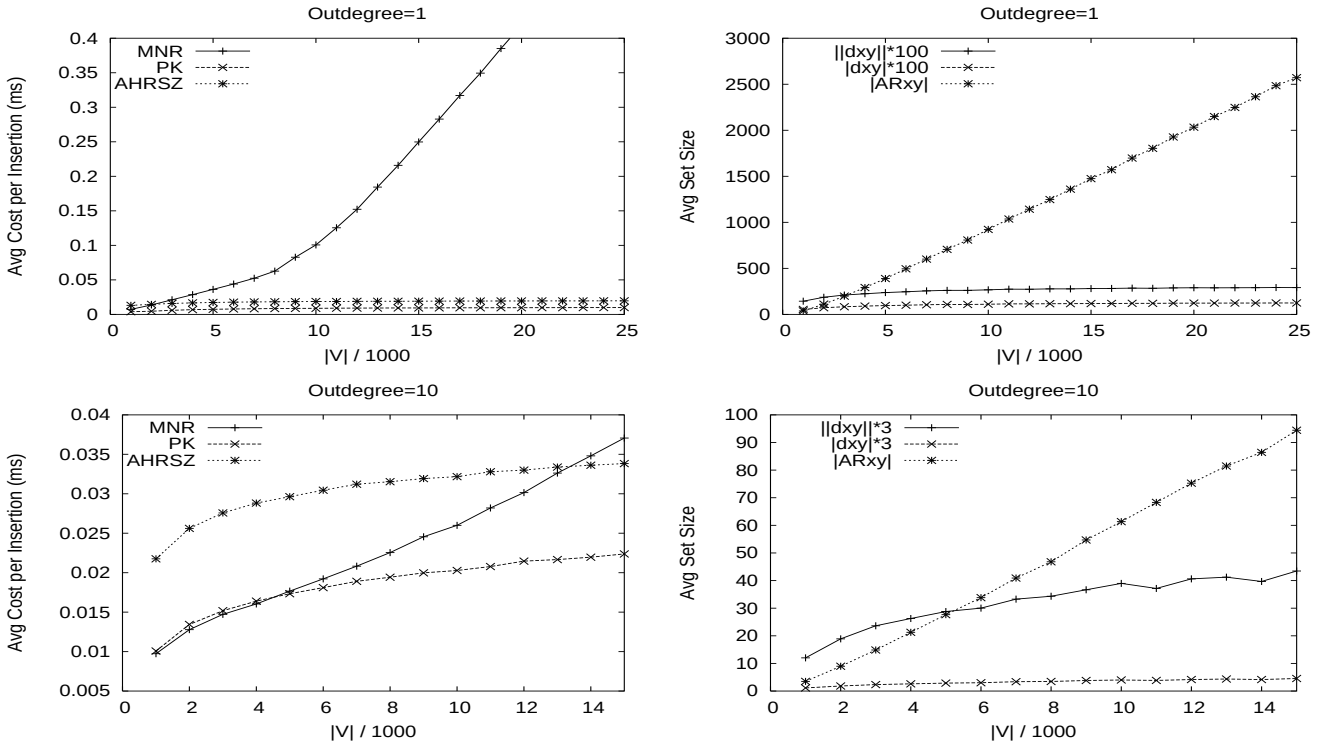


Figure 4: Experimental data on random graphs with varying —V—.

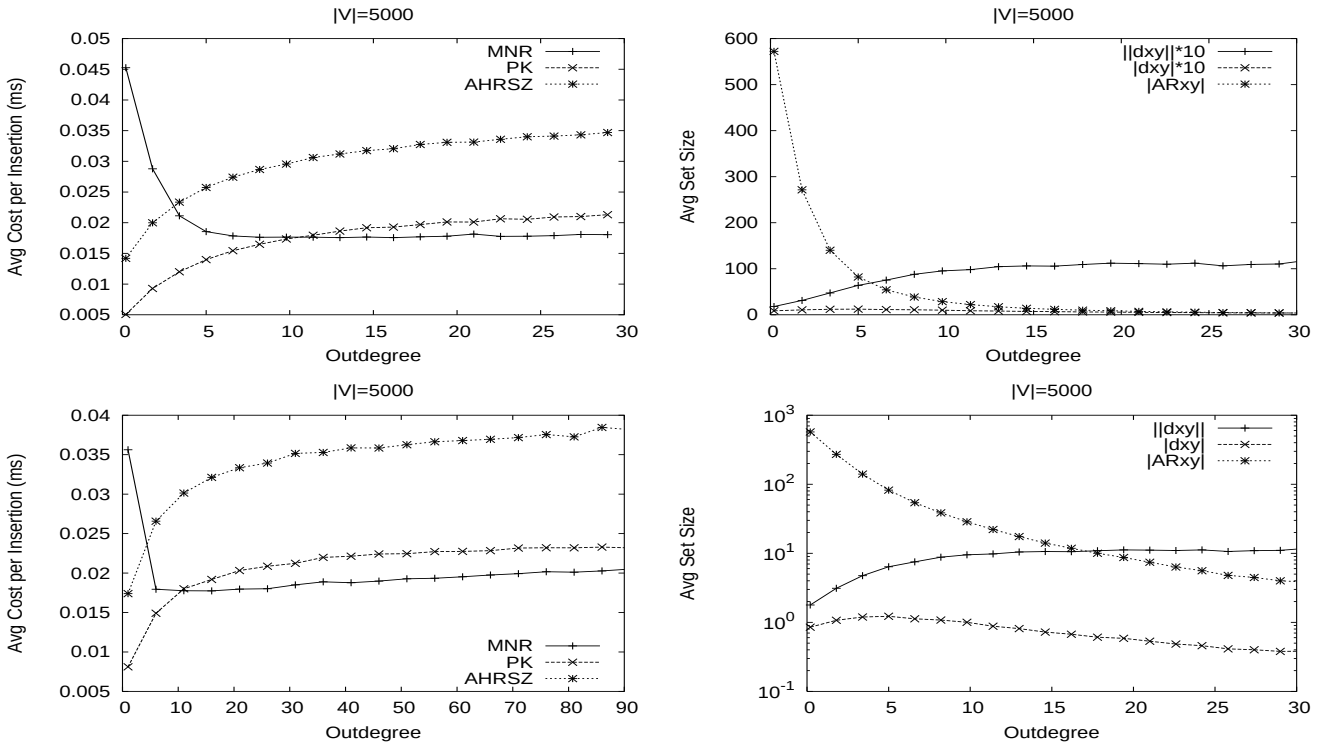


Figure 5: Experimental data for fixed sized graphs with varying outdegree.

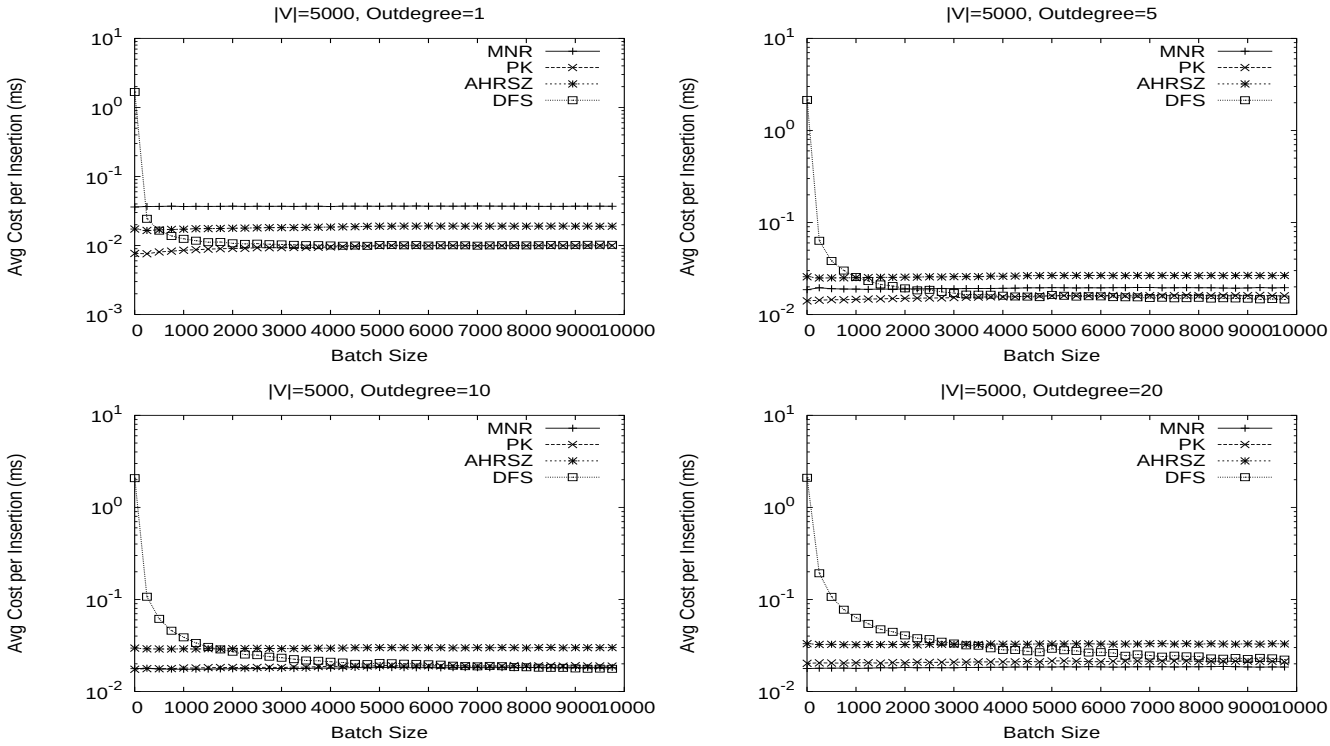


Figure 6: Experimental data for varying batch sizes comparing the three algorithms against a DFS based offline topological sort

propagate *in_component* information along edges traversed by the depth-first search. Figure 7 details how this can be done for the discovery stage of MNR. The extension to PK is much the same, although care must be taken to reset the visited flag for any nodes reached in `dfs-f` before moving onto `dfs-b` as all predecessors of nodes in the component must be found.

Once a cycle C has been detected we collapse all its nodes into one. Thus, all members of C are now represented by a single node in the ordering and we are effectively maintaining the topological order of the condensation graph.

7. CONCLUSION

We have presented a new algorithm for maintaining the topological order of a graph online, provided a complexity analysis, correctness proof and shown it performs better, for sparse graphs, than any previously known. Furthermore, we have provided the first empirical comparison of algorithms for this problem over a large number of randomly generated acyclic graphs. For the future, we are interested in investigating variants on these algorithms, which offer better performance for batch updates. We also consider a hybrid of MNR and PK and one of AHRSZ and MNR to be interesting ideas. 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.

8. REFERENCES

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, 2001.

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

[3] 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.

[4] 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.

[5] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, 1992.

[6] 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.

[7] 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.

[8] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[9] Esko Nuutila and Eljas Soisalon-Soininen. On finding the strongly connected components in a directed graph. *Information Processing Letters*, 49(1):9–14, January 1994.

[10] Harold N. Gabow. Path-based depth-first search for strong and biconnected components. *Information Processing Letters*, 74(3–4):107–114, 2000.

[11] M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proc. ACM Conference on Programming Language Design and Implementation*, pages 85–96, 1998.

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

[13] T. Reps. Optimal-time incremental semantic analysis for syntax-directed editors. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 169–176, 1982.

[14] A. M. Berman. *Lower And Upper Bounds For Incremental Algorithms*. PhD thesis, New Brunswick, New Jersey, 1992.

[15] G. Ramalingam. *Bounded incremental computation*, volume 1089 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[16] 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.

[17] 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.

[18] H. Djidjev, G. E. Pantziou, and C. D. Zaroliagis. Improved algorithms for dynamic shortest paths. *Algorithmica*, 28(4):367–389, 2000.

[19] 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.

[20] 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.

[21] 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.

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

[23] R. M. Karp. The transitive closure of a random digraph. *RSA: Random Structures & Algorithms*, 1(1):73–94, 1990.

[24] J. Siek, L. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.