Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Discrete Mathematics 25 (2006) 7–12
Electronic Notes in DISCRETE MATHEMATICS
www.elsevier.com/locate/endm

An $\mathcal{O}(n^{2.75})$ algorithm for online topological ordering [1]

Deepak Ajwani [a,2], Tobias Friedrich [a] and Ulrich Meyer [a]

[a] *Max-Planck-Institut für Informatik, Saarbrücken, Germany*

Abstract

We present a simple algorithm which maintains the topological order of a directed acyclic graph with n nodes under an online edge insertion sequence in $\mathcal{O}(n^{2.75})$ time, independent of the number of edges m inserted. For dense DAGs, this is an improvement over the previous best result of $\mathcal{O}(\min\{m^{\frac{3}{2}}\log n, m^{\frac{3}{2}} + n^2\log n\})$ by Katriel and Bodlaender.

Keywords: Topological ordering, Online algorithms, Graph algorithms.

1 Introduction

A topological order T of a given directed acyclic graph (DAG) $G = (V, E)$ (with $n := |V|$ and $m := |E|$) is a linear ordering of its nodes such that for all directed paths from $x \in V$ to $y \in V$ ($x \neq y$), it holds that $T(x) < T(y)$. There exist well known algorithms for computing the topological ordering of a DAG in $\mathcal{O}(m + n)$ in an offline setting (see e.g. [3]).

In the online variant of this problem, the edges of the DAG are not known in advance but are given one at a time. Each time an edge is added to the

[1] The full paper of this extended abstract has been accepted at SWAT 2006
[2] Research partially supported by DFG grant ME 2088/1-3.

1571-0653/$ – see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.endm.2006.06.050

DAG, we are required to update the bijective mapping T.

The online topological ordering has been studied in the following contexts

- As an online cycle detection routine in pointer analysis [9].
- Incremental evaluation of computational circuits [2].
- Compilation [5,7] where dependencies between modules are maintained to reduce the amount of recompilation performed when an update occurs.

The naïve way of maintaining an online topological order, i.e., to compute it each time from scratch with the offline algorithm, takes $\mathcal{O}(m^2 + mn)$ time. Marchetti-Spaccamela et al. [6] (MNR) gave an algorithm that can insert m edges in $\mathcal{O}(mn)$ time. Alpern et al. proposed a different algorithm [2] (AHRSZ) which runs in $\mathcal{O}(\|\delta\| \log \|\delta\|)$ time per edge insertion with $\|\delta\|$ measuring the number of edges of the minimal node subgraph that needs to be updated. Note that not all edges of this subgraph need to be visited and hence even $\mathcal{O}(\|\delta\|)$ time per insertion is not optimal. Katriel and Bodlaender (KB) [4] analyzed a variant of the AHRSZ algorithm and obtained an upper bound of $\mathcal{O}(\min\{m^{\frac{3}{2}} \log n, m^{\frac{3}{2}} + n^2 \log n\})$ for a general DAG. The algorithm by Pearce and Kelly [8] (PK) empirically outperforms the other algorithms for sparse random DAGs, although its worst-case runtime is inferior to KB.

We propose a simple algorithm that works in $\mathcal{O}(n^{2.75}\sqrt{\log n})$ time and $\mathcal{O}(n^2)$ space, thereby improving upon the results of Katriel and Bodlaender for dense DAGs. With some simple modifications in our data structure, we can get $\mathcal{O}(n^{2.75})$ time with $\mathcal{O}(n^{2.25})$ space or $\mathcal{O}(n^{2.75})$ *expected* time with $\mathcal{O}(n^2)$ space.

Our algorithm is dynamic, as it also supports deletion. However, our analysis holds only for a sequence of insertions. Our algorithm can also be used for online cycle detection in graphs, as well. Moreover, it permits an arbitrary starting point, which makes a hybrid approach possible, i.e., using the PK or KB algorithm for sparse graphs and ours for dense graphs.

2 Algorithm

We keep the current topological order as a bijective function, $T : V \rightarrow [1..n]$. If we start with an empty graph, we can initialize T with an arbitrary permutation, otherwise T is the topological order of the starting graph, computed offline. In this and the subsequent sections, we will use the following notations: $d(u, v)$ denotes $|T(u) - T(v)|$, $u < v$ is a short form of $T(u) < T(v)$, $u \rightarrow v$ denotes an edge from u to v, and $u \rightsquigarrow v$ expresses that v is reachable from u. Note that $u \rightsquigarrow u$, but *not* $u \rightarrow u$.

```
INSERT($u, v$)
    $\triangleright$ Insert edge $(u, v)$ and calculate new topological order
1   if $v \leq u$ then REORDER($u$,$v$)
2   insert edge $(u, v)$ in graph

REORDER($u, v$)
    $\triangleright$ Reorder nodes between $u$ and $v$ such that $v \leq u$
1   if $u = v$ then report detected cycle and quit
2   $A := \{w : v \to w$ and $w \leq u\}$
3   $B := \{w : w \to u$ and $v \leq w\}$
4   if $A = \emptyset$ and $B = \emptyset$
        then $\triangleright$ Correct the topological order
5            swap $u$ and $v$
6            update the data structure
        else $\triangleright$ Reorder node pairs between $u$ and $v$
7            for $v' \in \{v\} \cup A$ in decreasing topological order
8               for $u' \in B \cup \{u\} \wedge u' \geq v'$ in increasing topological order
9                   REORDER($u'$,$v'$)
```

Fig. 1. Our algorithm

Figure 1 gives the pseudo code of our algorithm. Throughout the process of inserting new edges, we maintain some data structures which are dependent on the current topological order. Inserting a new edge (u, v) is done by calling INSERT(u, v). If $v > u$, we do not change anything in the current topological order and simply insert the edge into the graph data structure. Otherwise, we call REORDER to update the topological order as well as the data structures dependent on it. Detection of $v = u$ indicates a cycle. If $v < u$, we first collect sorted sets A and B as defined in the code. If both A and B are empty, we swap the topological order of the two nodes and update the data structures. The query and the update operations are described in more detail along with our data structures in Section 2.1. Otherwise, we recursively call REORDER until everything inside is topologically ordered. To make these recursive calls efficient, we first merge the sorted sets $\{v\} \cup A$ and $B \cup \{u\}$ and using this merged list, compute the set $\{u' : (u' \in B \cup \{u\}) \wedge (u' > v')\}$ for each node $v' \in \{v\} \cup A$.

2.1 Data structure

We store the current topological order, as a set of two arrays, storing the bijective mapping T and its inverse. This ensures that finding $T(i)$ and $T^{-1}(u)$ are constant time operations.

The graph itself is stored as an array of vertices. For each vertex we maintain two adjacency lists, which keep the incoming and outgoing edges separately. Each adjacency list is stored as an array of buckets of vertices. The bucket can be kept as a balanced binary tree or as an array of n-bits or as a hash-table of a universal hashing function. Each bucket contains at most t nodes for a fixed t. Depending on the concrete implementation of the buckets, the parameter t is later chosen to be approximately $n^{0.75}$ so as to balance the number of inserts and deletes from the buckets and the extra edges touched by the algorithm. The i-th bucket ($i \geq 0$) of a node u contains all adjacent nodes v with $i \cdot t < d(u,v) \leq (i+1) \cdot t$. The nodes of a bucket are stored with node index (and not topological order) as their key. The bucket data structure should provide efficient support for the following three operations:

(i) Insert: Inserting an element in a given bucket.

(ii) Delete: Given an element and a bucket, find out if that element exists in that bucket. If yes, delete the element from there and return 1. Else, return 0.

(iii) Collect-all: Copying all the elements from the bucket to some vector.

We will now discuss how we do the insertion of an edge, computation of A and B, and updating the data-structure under swapping of nodes in terms of the above three basic operations.

Inserting an edge (u,v) means, inserting node v to the forward adjacency list of u and u to the backward adjacency list of v. This requires $\mathcal{O}(1)$ bucket inserts.

For given u and v, the set $A := \{w : v \to w \text{ and } w < u\}$ sorted according to the current topological order can be computed from the adjacency list of v by sorting all nodes of the first $\lceil d(u,v)/t \rceil$ outgoing buckets and choosing all w with $w < u$. This can be done by $\mathcal{O}(d(u,v)/t)$ collect-all operations on buckets collecting a total of $\mathcal{O}(|A| + t)$ elements. These elements are integers in the range $\{1 \,..\, n\}$ and can be sorted in $\mathcal{O}(|A| + t + \sqrt{n})$ time using a two-pass radix sort algorithm. The set B is computed likewise from the incoming edges.

When we swap two nodes u and v, we need to update the adjacency lists of u and v as well as that of all nodes w that are adjacent to u and/or v.

First, we show how to update the adjacency lists of u and v. If $d(u,v) > t$, we have to build their adjacency lists from scratch. Otherwise, the new bucket boundaries will differ from the old boundaries by $d(u,v)$ and at most $d(u,v)$ nodes will need to be transferred between any pair of consecutive buckets. The total number of transfers are therefore bounded by $d(u,v)\lceil n/t \rceil$. Determining whether a node should be transferred can be done in $\mathcal{O}(1)$ using the inverse mapping T^{-1} and as noted above, a transfer can be done in $\mathcal{O}(1)$ bucket inserts and deletes. Hence, updating the adjacency lists of u and v needs $\min\{n, d(u,v)\lceil n/t \rceil\}$ bucket inserts and deletes.

Let w be a node which is adjacent to u or v. Its adjacency list needs to be updated only if u and v are in different buckets. This corresponds to w being in different buckets of the adjacency lists of u and v. Therefore, the number of nodes to be transferred between different buckets for maintaining the adjacency lists of all w's is the same as the number of nodes that need to be transferred for maintaining the adjacency lists of u and v, i.e., $\min\{n, d(u,v)\lceil n/t \rceil\}$.

Updating the mappings T and T^{-1} after such a swap is trivial and can be done in constant time. Thus, we conclude that swapping nodes u and v can be done by $\mathcal{O}(d(u,v)\lceil n/t \rceil)$ bucket inserts and deletes.

The details of correctness proof and runtime analysis are available in [1].

3 Discussion

We have presented the first $o(n^3)$ algorithm for online topological ordering. The only non-trivial lower bound for this problem is by Ramalingam and Reps [10], who show that an adversary can force any algorithm maintaining explicit labels to need $\Omega(n \log n)$ time complexity for inserting $n-1$ edges. There is still a large gap between this, the trivial lower bound of $\Omega(m)$, and the upper bound of $\mathcal{O}(\min\{m^{1.5} + n^2 \log n, m^{1.5} \log n, n^{2.75}\})$. Bridging this gap remains an open problem.

References

[1] Ajwani, D., T. Friedrich and U. Meyer, *An $O(n^{2.75})$ algorithm for online topological ordering*, in: *10th Scandinavian Workshop on Algorithm Theory (SWAT)*, Lecture Notes in Computer Science **4059** (2006), pp. 53–64.

[2] Alpern, B., R. Hoover, B. K. Rosen, P. F. Sweeney and F. K. Zadeck, *Incremental evaluation of computational circuits*, in: *Proceedings of the first*

annual ACM-SIAM Symposium on Discrete Algorithms (1990), pp. 32–42.

[3] Cormen, T., C. Leiserson and R. Rivest, "Introduction to Algorithms," The MIT Press, Cambridge, MA, 1989, xvii + 1028 pp.

[4] Katriel, I. and H. L. Bodlaender, *Online topological ordering*, in: *Proceeding of the 16th ACM-SIAM Symposium on Discrete Algorithms (SODA), Vancouver*, 2005, pp. 443–450.

[5] Marchetti-Spaccamela, A., U. Nanni and H. Rohnert, *On-line graph algorithms for incremental compilation*, in: *Proceedings of International Workshop on Graph-Theoretic Concepts in Computer Science (WG 93)*, Lecture Notes in Computer Science **790**, 1993, pp. 70–86.

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

[7] Omohundro, S. M., C.-C. Lim and J. Bilmes, *The sather language compiler/debugger implementation*, Technical Report TR-92-017, International Computer Science Institute, Berkeley (1992).

[8] Pearce, D. J. and P. H. J. Kelly, *A dynamic algorithm for topologically sorting directed acyclic graphs*, in: *Proceedings of the Workshop on Efficient and experimental Algorithms*, Lecture Notes in Computer Science **3059** (2004), pp. 383–398.

[9] Pearce, D. J., P. H. J. Kelly and C. Hankin, *Online cycle detection and difference propagation for pointer analysis*, in: *Proceedings of the third international IEEE Workshop on Source Code Analysis and Manipulation (SCAM03)*, 2003.

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