I N F O R M A T I K

On Algorithms for Online Topological Ordering and Sorting

Irit Katriel

MPI–I–2004–1–003 February 2004

FORSCHUNGSBERICHT RESEARCH REPORT

M A X - P L A N C K - I N S T I T U T
F Ü R
I N F O R M A T I K

Stuhlsatzenhausweg 85 66123 Saarbrücken Germany

Author's Address

Irit Katriel
Stuhlsatzenhausweg 85
Max-Planck-Institut für Informatik,
66123 Saarbrücken, Germany
email: irit@mpi-sb.mpg.de

Abstract

We derive several results about algorithms for the online topological ordering and sorting problems. The main result is that a slight variation on an algorithm by Alpern et al. yields an amortized complexity of $\sqrt{m}\log n$ per edge for an n-node m-edge DAG. This is an improvement, for all except very dense graphs, over the best known previous result of $O(n)$ per edge due to Marchetti-Spaccamela et al.

Keywords

Topological Order, Topological Sorting, Online Algorithm, Amortized Analysis, Lower Bound

1 Introduction

Given a directed acyclic graph (DAG) $G = (V, E)$, the *topological ordering* (TO) problem is to assign a value $ord(v)$ to every $v \in V$ such that $(u, v) \in E$ implies $ord(u) < ord(v)$. The *topological sorting* problem is the same, with the additional requirements that for all $v \in V$, $1 \leq ord(v) \leq |V|$ and for any $u, v \in V$, $u \neq v$ implies $ord(u) \neq ord(v)$. Both problems can be solved in linear time [9].

We are interested in the online variants of these problems, where the edges of the DAG are not known in advance but are given one at a time and we are required to update the mapping *ord* every time an edge is added to the DAG. Assume that we have a DAG and a valid topological ordering (sorting) function *ord*. Assume that we add an edge (x, y) to this DAG. If $ord(x) < ord(y)$ then *ord* is a valid topological ordering (sorting) function for the new DAG and no updating is necessary. Otherwise, we say that the new edge (x, y) violates the topological order *ord*. In this case, we need to find a new function ord' which is a valid topological ordering (sorting) function for the new DAG.

The online TO problem was studied in the context of compilation [5] where dependencies between modules are maintained to reduce the amount of recompilation performed when an update occurs, source code analysis [6] where the aim is to statically determine the smallest possible target set for all pointers in a program and as a subroutine of an algorithm for incremental evaluation of computational circuits [1].

Finding efficient solutions for the TS problem seems to be of theoretical more than practical interest. However, some of the algorithms that were proposed for the TO problem, in particular the ones that use simpler data structures, actually solve the more restricted TS problem. This means that lower bounds for the TS problem are of interest because they tell us what is the most that we can expect to get from an online TS algorithm.

1.1 Known results

The trivial solution for the online problem is to compute a new topological order from scratch whenever an edge is inserted. Since this takes time $\Theta(n + m)$ for an n-node m-edge DAG, the complexity of inserting m edges is $\Theta(m^2)$[1].

Marchetti-Spaccamela et al. [4] gave an algorithm for the TS problem that can insert m edges in $O(mn)$ time, giving an amortized time bound of

[1]Here and in the rest of the paper we assume $n \leq m \leq n(n-1)/2$.

$O(n)$ per edge instead of the trivial $O(m)$. This is the best amortized result known so far. Pearce and Kelly [6] propose a different algorithm and show experimentally that it achieves a speedup on sparse inputs.

Alpern et al. [1] designed an algorithm for the online TO problem which runs in time $O(\|\delta\| \log \|\delta\|)$ in the *bounded incremental computation* model [8]. In this model, the parameter $\|\delta\|$ measures, upon an edge insertion, the size (number of nodes and edges) in a minimal subgraph that we need to update in order to obtain a valid topological order ord' for the graph after the edge insertion. Since $\|\delta\|$ can be anywhere between 0 and half of the number of edges in the graph, the bounded complexity result does not provide much information about the cost of a sequence of updates. All it guarantees is that each individual update could not have been performed much faster.

1.2 Our results

This paper includes several results about the above mentioned algorithms. The first, and most interesting, is an analysis that shows that a slight variation on the algorithm by Alpern et al. needs $O(m^{3/2} \log n)$ time to handle m edge insertions. This implies an amortized time of $O(\sqrt{m} \log n)$ per edge and is an improvement over the previous result whenever $m = O((\frac{n}{\log n})^2)$. I.e., for all except very dense graphs.

Next, we show that the complexity analysis of the algorithm by Marchetti-Spaccamela et al. is tight. That is, for any $n \leq m \leq n(n-1)$ there is a sequence of m edge insertions which will take the algorithm $\Omega(mn)$ time to process.

Finally, we define a class of algorithms that includes all known solutions (except for the trivial one of recomputing the solution from scratch) and show that any algorithm in this class for the TS problem has to perform $\Omega(n^2)$ node relabellings in the worst case. We show that the algorithm by Pearce and Kelly never performs more than n^2 node relabellings, and is therefore worst-case optimal.

2 An $O(\sqrt{m} \log n)$ amortized upper bound

Figure 1 shows a slight variation on the algorithm by Alpern et al. [1], which works as follows. The *ord* labels of the nodes are maintained by an *Order Data Structure* [2, 3] *ORD*, which is a data structure that allows to maintain a total order over a list of items and to perform the following operations in constant amortized time: $Insert(x)$ inserts the item x in an arbitrary location in the total order. $InsertBefore(x, y)$ ($InsertAfter(x, y)$) inserts the item x immediately before (after) the item y in the total order, $Delete(x)$ removes the item x and the query $Order(x, y)$ determines whether x precedes y or y precedes x in the total order. These operations are implemented by assigning integer labels to the items in the list such that the label assigned to x is smaller than the label assigned to y iff x precedes y in the total order.

Initially, no edges exist in the graph so the nodes are inserted to ORD in an arbitrary order. Then, whenever an edge $(Source, Target)$ is inserted into the graph, the function $AddEdge$ is called. When this function returns, the total order ORD is a valid topological order for the modified graph.

It remains to describe how the algorithm determines which nodes need to be relabelled and how they are relabelled. Given a total order ord on the nodes and the edge $(Source, Target)$ which is to be inserted, we define the *affected region* AR to be the set of nodes $\{v | ord(Target) \leq ord(v) \leq ord(Source)\}$. The set $FromTarget$ contains all the nodes from AR which are reachable from $Target$ by a path in the DAG and the set $ToSource$ is the set of nodes in AR from which $Source$ can be reached by a path.

The algorithm begins to explore the nodes in $FromTarget$ and $ToSource$, the first by beginning at $Target$ and advancing by order of the ord labels of the nodes, and the second by beginning at $Source$ and advancing in reverse order of the ord labels. It builds two sets of nodes, $ToS \subseteq ToSource$ and $FromT \subseteq FromTarget$, such that the desired order ord' can be obtained by relabelling the nodes in $ToS \cup FromT$. Two priority queues are used for this search: $SourceQueue$ for the nodes that are candidates for insertion into ToS and $TargetQueue$ for the nodes that are candidates for insertion into $FromT$. In each iteration, there is one node s which is a candidate for insertion into ToS (the node with maximal ord label which reaches a node in ToS but is not in ToS) and one node t which is a candidate for insertion into $FromT$ (the node with minimal ord label which can be reached from a node in $FromT$ but is not in $FromT$). The algorithm adds at least one of them (possibly both) to the relevant set. The way in which it decides which candidate(s) to add aims to balance the number of edges outgoing from nodes in $FromT$ and the number of edges incoming into nodes in ToS. For each node $v \in V$, let $InDegree[v]$ be the number of edges incoming into v and $OutDegree[v]$ be the number of edges outgoing from v. Once a node becomes a candidate for insertion to ToS or $FromT$, it remains the candidate for insertion into its set until it is inserted or the function $AddEdge$ terminates. The algorithm maintains two counters, one for the current s and the other for the current t. When a node v becomes a candidate for insertion into ToS ($FromT$), the relevant counter is set to $InDegree[v]$ ($OutDegree[v]$). Whenever the two candidates are considered, the value of the smaller counter is subtracted from both counters. Then, each candidate is inserted into its respective set if the value of its counter is 0.

The construction of the sets *ToS* and *FromT* ends when $ord(s) < ord(t)$. Then, the nodes in *ToS* are deleted from *ORD* and are reinserted, in the same relative order among themselves, immediately after s. Similarly, the nodes in *FromT* are deleted and reinserted immediately before t. Finally, the edge (*Source*, *Target*) is inserted to the DAG.

Remark. We have mentioned above that the algorithm we describe is a variation on the one by Alpern et al. The change we made is that the *SourceDegree* and *TargetDegree* counters are initialized to the indegree or outdegree of a node. Alpern et al. initialize these values to the total number of edges incident on the node. In the bounded complexity model, their definition is appropriate because it is the sum of the total degrees that should be minimized. For our analysis, however, we need to count only edges that influence the change in potential due to the operation.

2.1 Complexity analysis

Let $E = \{e_1, \ldots, e_m\}$ be the edges of the DAG, sorted by the order in which they are inserted. After inserting $e_1, \ldots, e_k$, for any $1 \leq i < j \leq m$ we say that the pair of edges $e_i = (x_i, y_i)$, $e_j = (x_j, y_j)$ is *ordered* if $j \leq k$ and there is a path from y_i to x_j or from y_j to x_i. That is, if both edges have been inserted to the DAG and there is a path that determines their relative positions in the topological order. Otherwise, we say that the edges are *unordered*.

We define a potential function Φ = Number of unordered pairs of edges. Initially, all pairs of edges are unordered, so $\Phi_0 = m(m-1)/2 < m^2$.

Upon the insertion of an edge, let n_t (n_s) be the number of nodes in *FromT* (*ToS*). Let m_t (m_s) be the number of edges outgoing from (incoming to) nodes in *FromT* (*ToS*). Finally, let ℓ_t (ℓ_s) be the number of edges outgoing from (incoming to) the node that was candidate for insertion into *FromT* (*ToS*) when the algorithm terminated. If this node does not exist, i.e., *TargetQueue* (*SourceQueue*) is empty at the end, then $\ell_t = 0$ ($\ell_s = 0$).

Lemma 1 *The algorithm spends on an insertion* $O(1 + \max\{m_s, m_t\} \log n)$ *time.*

Proof The time that the algorithm spends on the insertion is a constant amount c, plus $O(m_t \log m_t + m_s \log m_s)$ to construct the sets *ToS* and *FromT* and $O(n_t + n_s)$ for the relabelling. With $n_i \leq m_i$ for $i = 1, 2$, we have that the time spent is $O(1 + m_t \log m_t + m_s \log m_s) = O(1 + (m_t + m_s) \log m) = O(1 + \max\{m_s, m_t\} \log n)$. ∎

```
Function AddEdge (Source , Target )
    SourceQueue ← []; TargetQueue ← [] (* Empty priority queues *)
    SourceStack ← [] ; TargetStack ← [] (* Empty stacks *)
    s ← Source ; t ← Target
    SourceDegree ← InDegree[s] ; TargetDegree ← OutDegree[t]

    (* Discovery phase: decide which nodes should be relabelled. *)
    while (ord(s) > ord(t)) do
        d ← min{SourceDegree, TargetDegree}
        SourceDegree ← SourceDegree − d
        TargetDegree ← TargetDegree − d
        if SourceDegree = 0 then
            foreach (w, s) ∈ E do SourceQueue.Insert(w)
            SourceStack.Push(s)
            s ← SourceQueue.ExtractMax
            SourceDegree ← InDegree[s]
        endif
        if TargetDegree = 0 then
            foreach (t, w) ∈ E do TargetQueue.Insert(w)
            TargetStack.Push(t)
            t ← TargetQueue.ExtractMin
            TargetDegree ← OutDegree[t]
        endif
    end while

    (* Relabel the nodes *)
    while SourceStack.NotEmpty do
        s' ← SourceStack.Pop
        ORD.Delete(s'); ORD.InsertAfter(s', s)
    end while
    while TargetStack.NotEmpty do
        t' ← TargetStack.Pop
        ORD.Delete(t'); ORD.InsertBefore(t', t)
    end while

    (* Insert the edge *)
    E ← E ∪ {(Source, Target)}
    InDegree[Target] ← InDegree[Target] + 1
    OutDegree[Source] ← OutDegree[Source] + 1
end
```

Figure 1: The algorithm by Alpern et al.

Lemma 2 *For every edge insertion, (1) $m_t \leq m_s + \ell_s$ and (2) $m_s \leq m_t + \ell_t$.*

Proof We show by induction on the number of iterations of the while loop which constructs the sets *FromT* and *ToS*, that after each iteration, $m_t + \ell_t - \mathit{TargetDegree} = m_s + \ell_s - \mathit{SourceDegree}$. Since at all times $0 \leq \mathit{TargetDegree} \leq \ell_t$ and $0 \leq \mathit{SourceDegree} \leq \ell_s$, (1) and (2) follow for the final as well as the intermediate values of m_s, ℓ_s, m_t, ℓ_t.

Initially, $m_t = m_s = 0$, $\mathit{TargetDegree} = \ell_t$ and $\mathit{SourceDegree} = \ell_s$, so the equality holds. Assume that it holds up to iteration $i-1$ of the discovery loop. We will show that it holds also after iteration i. For any value $v \in \{m_s, \ell_s, m_t, \ell_t, \mathit{SourceDegree}, \mathit{TargetDegree}\}$, let v denote this value after iteration $i-1$ and v' the same value after iteration i.

Since the algorithm did not terminate after iteration $i-1$, we know that there is a candidate for insertion into each of *FromT* and *ToS* at the beginning of iteration i. Assume w.l.o.g. that $\mathit{SourceDegree} \leq \mathit{TargetDegree}$. Then s will be inserted into *ToS*, so $m'_s = m_s + \ell_s$ and $\mathit{SourceDegree}' = \ell'_s$. We get that $m'_s + \ell'_s - \mathit{SourceDegree}' = m_s + \ell_s$.

For t, there are two cases. The first case is that $\mathit{TargetDegree} = \mathit{SourceDegree}$ so t will be inserted to *FromT*. Similar to the above, we get that $m'_t + \ell'_t - \mathit{TargetDegree}' = m_t + \ell_t$. Since $\mathit{SourceDegree} = \mathit{TargetDegree}$, we know from the induction hypothesis that $m_t + \ell_t = m_s + \ell_s$, which means that $m'_s + \ell'_s - \mathit{SourceDegree}' = m'_t + \ell'_t - \mathit{TargetDegree}'$.

The second case is that $\mathit{TargetDegree} < \mathit{SourceDegree}$ so t will not be inserted into *FromT*. This implies that $m'_t = m_t$, $\ell'_t = \ell_t$ and $\mathit{TargetDegree}' = \mathit{TargetDegree} - \mathit{SourceDegree}$. We get that $m'_t + \ell'_t - \mathit{TargetDegree}' = m_t + \ell_t + \mathit{TargetDegree} - \mathit{SourceDegree}$. By the induction hypothesis, this is equal to $m_s + \ell_s$, which is equal to $m'_s + \ell'_s - \mathit{SourceDegree}'$. ∎

For $0 \leq i \leq m$, let Φ_i be the value of the potential function after the insertion of $e_1, \ldots, e_i$ to the graph. For $1 \leq i \leq m$, let $\Delta_i = \Phi_i - \Phi_{i-1}$ be the change in potential due to the insertion of e_i.

Lemma 3 *When the algorithm handles an insertion, $\Delta_i \leq -\max\{m_t^2, m_s^2\}$. That is, the potential decreases by at least $\max\{m_t^2, m_s^2\}$.*

Proof Let $e_t = (x_t, y_t)$ be an edge outgoing from a node in *FromT* or the last candidate for insertion into *FromT* and $e_s = (x_s, y_s)$ be an edge incoming into a node in *ToS* or the last candidate for insertion into *ToS*. We will show that e_t and e_s were unordered before the insertion and ordered after the insertion. After the insertion, y_t is reachable from *Target* by a path P_t that uses e_t and *Source* is reachable from x_s by a path P_s that uses e_s.

Concatenating the paths with the inserted edge between them gives the path $P_s \circ (\mathit{Source}, \mathit{Target}) \circ P_t$ from x_s to y_t that uses both edges. Hence, the edges are ordered such that $\mathit{ord}'(y_s) < \mathit{ord}'(x_t)$.

Assume that they are ordered before the insertion as well. If they are ordered with $\mathit{ord}(y_t) < \mathit{ord}(x_s)$ then the insertion introduces a cycle, so the graph is not a DAG. So they are ordered such that $\mathit{ord}(y_s) < \mathit{ord}(x_t)$. If y_s and x_t were candidates for insertion to their sets in the same iteration of the discovery phase of the algorithm, then their relative order in ORD would imply that the algorithm terminates without adding either one of them. Assume w.l.o.g. that y_s was inserted before x_t became the candidate. Then if x_t became the candidate for insertion into FromT, the candidate for insertion into ToS was a node v such that $\mathit{ord}(v) < \mathit{ord}(y_s) < \mathit{ord}(x_t)$. So the algorithm will terminate without adding x_t. On the other hand, if x_t never became the candidate for insertion, then it obviously was not inserted.

We get that the potential decreased by at least $(m_s + \ell_s)(m_t + \ell_t)$, which by Lemma 2 is at least $\max\{m_s^2, m_t^2\}$. ∎

Theorem 1 *The algorithm in Figure 1 needs $O(m^{3/2} \log n)$ time to insert m edges to an initially empty n-node graph.*

Proof By Lemma 1, the algorithm spends $O(1 + \max\{m_s, m_t\} \log n)$ time on an insertion, while by Lemma 3, the potential decreases by at least $\max\{m_s^2, m_t^2\}$.

For $i = 1, \ldots, m$, let x_i be the value of $\max\{m_s, m_t\}$ upon the insertion of the ith edge. We get that the total time spent on the m insertions is $O(\sum_{i=1}^m (1 + x_i \log n)) = O(m + \sum_{i=1}^m x_i \log n)$ and the potential decreased by a total of $\sum_{i=1}^m \Delta_i \geq \sum_{i=1}^m x_i^2$. Since the potential never increases, is initially less than m^2 and is never negative, we have $\sum_{i=1}^m x_i^2 \leq m^2$. Using Cauchy's inequality, we get that $\sum_{i=1}^m x_i \leq m^{3/2}$, so the total time spent is $O(m + \sum_{i=1}^m x_i \log n) = O(m^{3/2} \log n)$. ∎

2.2 This analysis is almost tight

We now show that our analysis is almost tight. That is, we show that for each $n \leq m \leq n(n-1)/2$ there is an input with n nodes and m edges that will take the algorithm by Alpern et al. $\Omega(m^{3/2})$ time to process, which is merely a log factor away from the upper bound.

Let $k = \sqrt{m}/4$ and let $\{v_1, \ldots, v_n\}$ be the nodes of the DAG sorted by their initial ord order. The first part of the input consists of the edge (v_i, v_j)

for all $i \in \{1, \ldots, k\}, j \in \{2k+1, \ldots, 3k\}$ and $i \in \{k+1, \ldots, 2k\}, j \in \{3k+1, \ldots, 4k\}$.

In the second part, an edge is introduced from v_i to v_j for each $i \in \{2k+1, \ldots, 3k\}$ and $j \in \{k+1, \ldots, 2k\}$, in the following order: For all $i \in \{2k+1, \ldots, 3k-1\}$, all edges outgoing from v_i appear before all edges outgoing from v_{i+1} and for all $j \in \{k+2, \ldots, 2k\}$, the edge (v_i, v_j) appears before the edge (v_i, v_{j-1}).

So far, the input consisted of $3k^2 = 3m/16$ edges. In the third part, we complete this to m by inserting any $13m/16$ edges that are not already in the DAG.

We show that the second part will cause the algorithm to perform $\Omega(m^{3/2})$ work. Every edge (v_i, v_j) that is inserted in the second part is from a node v_i to a node v_j which is immediately before it in the topological order. Since $\mathit{InDegree}[v_i] \geq k$ and $\mathit{OutDegree}[v_j] \geq k$, the algorithm will insert v_i into ToS or v_j into FromT (possible both) and will traverse at least k incident edges. Thus, it will spend $\Omega(k)$ for each edge and since the number of edges inserted in the second part is k^2, we get a total complexity of $\Omega(k^3) = \Omega(m^{3/2})$.

3 Amortized lower bound for the algorithm by Marchetti-Spaccamela et al.

The amortized bound of $O(n)$ per edge insertion was obtained by Marchetti-Spaccamela et al. [4] with an algorithm that solves the TS problem. This algorithm places the nodes in an array of size n such that the index of a node in the array is its *ord* label. Initially, the nodes are placed in arbitrary order in the array. When an edge $(\mathit{Source}, \mathit{Target})$ is inserted such that $\mathit{ord}(\mathit{Source}) > \mathit{ord}(\mathit{Target})$, some nodes need to be relabelled, as before. This time, all nodes in the affected region AR are relabelled. This is done in two phases: In the first, the set *FromTarget* is identified by a search beginning at *Target*. In the second, the nodes in AR are permuted such that the nodes in *FromTarget* appear after all other nodes in AR. The time for the second phase is $O(n)$, so the running time is dominated by the time for the first phase, which is proportional to the number of edges in the subgraph that is traversed. If an edge (u, v) is traversed because of the insertion of an edge (x, y), then from that point in time, $\mathit{ord}(x) < \mathit{ord}(u)$, so the edge (u, v) will not be explored due to an insertion of another edge whose source node is x. This means that each edge is traversed at most n times so the total time for traversing the edges is $O(mn)$.

In this section we show that this upper bound is tight. That is, we show that for any $n \leq m \leq n(n-1)/2$, there is an input on which this algorithm performs $\Omega(mn)$ work for inserting m edges into an initially empty n-node graph.

We begin with n nodes and no edges. The nodes are labelled 1 to n in some arbitrary order. Let $v_1, \ldots, v_n$ be the nodes by increasing order of their labels at the beginning. The adversary constructs the following input:

Let $k = \frac{\sqrt{m}}{4}$. Phase 1: Create a complete bipartite graph with k nodes on each side, by inserting an edge (v_i, v_j) for each $i \in \{1, \ldots, k\}$ and for each $j \in \{k+1, \ldots, 2k\}$.

Phase 2: Add an edge from v_{2k+1} to each of $\{v_1, \ldots, v_k\}$.

Phase 3: Add the edges (v_{i+1}, v_i) for $i = 2k+1, \ldots, n-1$.

In phase 3, we insert $n-2k-1 = n-\frac{\sqrt{m}}{2}-1 \geq \frac{n}{2}-1 = \Omega(n)$ edges. When the edge (v_{i+1}, v_i) is inserted, all nodes $v_1, \ldots, v_{i-1}$ are reachable from v_i, so the subgraph induced by them is traversed, including the k^2 edges between $\{v_1, \ldots, v_k\}$ and $\{v_{k+1}, \ldots, v_{2k}\}$. We get that in phase 3 the algorithm spends $\Omega(nk^2) = \Omega(mn)$ time.

4 Node relabellings for topological sorting

Two algorithms are known for topological sorting: The algorithm by Marchetti-Spaccamela et al. which was described in the previous section and an algorithm by Pearce and Kelly [6] which will be described below. Both of these algorithms have the following property: they change labels only of nodes inside the affected region AR. We call an algorithm with this property a *local* algorithm. We can prove a lower bound for local algorithms for the TS problem:

Lemma 4 *Any local algorithm for the topological sorting problem needs to perform $\Omega(n^2)$ relabellings in the worst case, when $m \geq n$ edges are inserted to an initially empty graph.*

Proof Initially we have n isolated nodes which are arbitrarily labelled with distinct labels in $\{1, \ldots, n\}$. As in the previous section, let $v_1, \ldots, v_n$ be the nodes by increasing order of their labels at the beginning. The adversary builds a chain through them by adding the $n-1$ edges (v_{i+1}, v_i) for i from 1 to $n-1$. After inserting i edges, we have a chain of $i+1$ nodes whose *ord* values are in $\{1, \ldots, i+1\}$ and $n-i-1$ isolated nodes whose *ord* values are in $\{i+2, \ldots, n\}$. Thus, each insertion causes all nodes of the chain to be relabelled, so the total number of relabellings is $\sum_{i=1}^{n-1} i+1 = \Omega(n^2)$. ∎

The algorithm by Pearce and Kelly [6] relabels only a subset of AR: Those nodes in AR which are reachable from *Target* or can reach *Source*. In the worst case (e.g., the input used in the proof of Lemma 4), the number of nodes relabelled is the same as in the algorithm by Marchetti-Spaccamela et al. However, in other cases this number may be much smaller, and experimental results show that on random sparse graphs this algorithm is faster [6]. The following lemma proves that it is worst-case optimal in terms of the number of node relabellings it performs.

Lemma 5 *For any $m \leq n(n-1)/2$, the algorithm by Pearce and Kelly performs at most n^2 node relabellings when inserting m edges to an initially empty n-node graph.*

Proof We define a potential function which resembles the one we used in Section 2, but counts pairs of unordered nodes instead of edges. We say that a pair of nodes u, v is *ordered* if there is a path in the DAG from u to v or from v to u and *unordered* otherwise.

The potential function is $\Psi = |\{< u, v > | u, v \text{ are unordered}\}|$. That is, the number of pairs of nodes such that there does not exist a path in the DAG from either one of them to the other. Initially, $\Psi = n(n-1) < n^2$.

Upon the insertion of an edge $(Source, Target)$ to the graph, the nodes that are relabelled are the nodes in the two sets *FromTarget* and *ToSource*. For each $v \in FromTarget$, the pair $< Source, v >$ was unordered before the insertion and is ordered after the insertion. Similarly, for each $v \in ToSource$, the pair $< Target, v >$ was unordered before the insertion and is ordered after the insertion. We get that the algorithm relabels $O(|FromTarget| + |ToSource|)$ nodes and this decreases the potential by $\Omega(|FromTarget| + |ToSource|)$. Since $\Psi \geq 0$ and it cannot increase when we perform only insertions, we get that the total number of node relabellings does not exceed n^2. ∎

5 Conclusion

We have shown that the online topological ordering problem can be solved in $\sqrt{m} \log n$ amortized time per edge, an improvement over the previous result for all except very dense graphs. We have also shown a lower bound of $\Omega(n^2)$ node relabellings, and hence also time, for any local algorithm that solves the more restricted problem of topological sorting. For sparse graphs, with $m = O(n)$, this lower bound for topological sorting is higher than the upper bound we have shown for topological ordering: $O(n^{3/2} \log n)$. This means that it is not possible to beat the upper bound we have shown with a local algorithm that solves the TS problem.

The only non-trivial lower bound on the time required to solve the online topological ordering problem was proved by Ramalingam and Rep [7], who show that an adversary can force any algorithm to perform $\Omega(n \log n)$ node relabelling operations while inserting $n-1$ edges (creating a chain). The upper bound we have shown in this paper for $m = n-1$ is $O(n^{3/2} \log n)$. Thus, the gap is still large.

I wish to thank David J. Pearce and Seth Pettie for helpful discussions.

References

[1] B. Alpern, R. Hoover, B. K. Rosen, P. F. Sweeney, and F. Kenneth Zadeck. Incremental evaluation of computational circuits. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 32–42. Society for Industrial and Applied Mathematics, 1990.

[2] 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 10th Annual European Symposium on Algorithms*, pages 152–164. Springer-Verlag, 2002.

[3] P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the nineteenth annual ACM conference on Theory of computing*, pages 365–372. ACM Press, 1987.

[4] A. Marchetti-Spaccamela, U. Nanni, and H. Rohnert. Maintaining a topological order under edge insertions. *Inf. Process. Lett.*, 59(1):53–58, 1996.

[5] S.M. Omohundro, C. Lim, and J. Bilmes. The Sather language compiler/debugger implementation. Technical Report TR-92-017, International Computer Science Institute, Berkeley, March 1992.

[6] D. J. Pearce and P. H. J. Kelly. A dynamic algorithm for topologically sorting directed acyclic graphs (submitted to Workshop on Efficient and experimental Algorithms). Technical report, Imperial College, 2004.

[7] G. Ramalingam and T. Reps. On competitive on-line algorithms for the dynamic priority-ordering problem. *Inf. Process. Lett.*, 51(3):155–161, 1994.

[8] G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *Theor. Comput. Sci.*, 158(1-2):233–277, 1996.

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

mpi

INFORMATIK

Below you find a list of the most recent technical reports of the Max-Planck-Institut für Informatik. They are available by anonymous ftp from `ftp.mpi-sb.mpg.de` under the directory `pub/papers/reports`. Most of the reports are also accessible via WWW using the URL `http://www.mpi-sb.mpg.de`. If you have any questions concerning ftp or WWW access, please contact `reports@mpi-sb.mpg.de`. Paper copies (which are not necessarily free of charge) can be ordered either by regular mail or by e-mail at the address below.

Max-Planck-Institut für Informatik
Library
attn. Anja Becker
Stuhlsatzenhausweg 85
66123 Saarbrücken
GERMANY
e-mail: `library@mpi-sb.mpg.de`

MPI-I-2004-1-003	I. Katriel	On Algorithms for Online Topological Ordering and Sorting
MPI-I-2004-1-002	P. Sanders, S. Pettie	A Simpler Linear Time 2/3 - epsilon Approximation for Maximum Weight Matching
MPI-I-2004-1-001	N. Beldiceanu, I. Katriel, S. Thiel	Filtering algorithms for the Same and UsedBy constraints
MPI-I-2003-NWG2-002	F. Eisenbrand	Fast integer programming in fixed dimension
MPI-I-2003-NWG2-001	L.S. Chandran, C.R. Subramanian	Girth and Treewidth
MPI-I-2003-4-009	N. Zakaria	FaceSketch: An Interface for Sketching and Coloring Cartoon Faces
MPI-I-2003-4-008	C. Roessl, I. Ivrissimtzis, H. Seidel	Tree-based triangle mesh connectivity encoding
MPI-I-2003-4-007	I. Ivrissimtzis, W. Jeong, H. Seidel	Neural Meshes: Statistical Learning Methods in Surface Reconstruction
MPI-I-2003-4-006	C. Roessl, F. Zeilfelder, G. Nürnberger, H. Seidel	Visualization of Volume Data with Quadratic Super Splines
MPI-I-2003-4-005	T. Hangelbroek, G. Nürnberger, C. Roessl, H.S. Seidel, F. Zeilfelder	The Dimension of C^1 Splines of Arbitrary Degree on a Tetrahedral Partition
MPI-I-2003-4-004	P. Bekaert, P. Slusallek, R. Cools, V. Havran, H. Seidel	A custom designed density estimation method for light transport
MPI-I-2003-4-003	R. Zayer, C. Roessl, H. Seidel	Convex Boundary Angle Based Flattening
MPI-I-2003-4-002	C. Theobalt, M. Li, M. Magnor, H. Seidel	A Flexible and Versatile Studio for Synchronized Multi-view Video Recording
MPI-I-2003-4-001	M. Tarini, H.P.A. Lensch, M. Goesele, H. Seidel	3D Acquisition of Mirroring Objects
MPI-I-2003-2-004	A. Podelski, A. Rybalchenko	Software Model Checking of Liveness Properties via Transition Invariants
MPI-I-2003-2-003	Y. Kazakov, H. Nivelle	Subsumption of concepts in $DL\ \mathcal{FL}_0$ for (cyclic) terminologies with respect to descriptive semantics is PSPACE-complete
MPI-I-2003-2-002	M. Jaeger	A Representation Theorem and Applications to Measure Selection and Noninformative Priors
MPI-I-2003-2-001	P. Maier	Compositional Circular Assume-Guarantee Rules Cannot Be Sound And Complete
MPI-I-2003-1-018	G. Schaefer	A Note on the Smoothed Complexity of the Single-Source Shortest Path Problem

MPI-I-2003-1-017	G. Schäfer, S. Leonardi	Cross-Monotonic Cost Sharing Methods for Connected Facility Location Games
MPI-I-2003-1-016	G. Schäfer, N. Sivadasan	Topology Matters: Smoothed Competitive Analysis of Metrical Task Systems
MPI-I-2003-1-015	A. Kovács	Sum-Multicoloring on Paths
MPI-I-2003-1-014	G. Schäfer, L. Becchetti, S. Leonardi, A. Marchetti-Spaccamela, T. Vredeveld	Average Case and Smoothed Competitive Analysis of the Multi-Level Feedback Algorithm
MPI-I-2003-1-013	I. Katriel, S. Thiel	Fast Bound Consistency for the Global Cardinality Constraint
MPI-I-2003-1-012		- not published -
MPI-I-2003-1-011	P. Krysta, A. Czumaj, B. Voecking	Selfish Traffic Allocation for Server Farms
MPI-I-2003-1-010	H. Tamaki	A linear time heuristic for the branch-decomposition of planar graphs
MPI-I-2003-1-009	B. Csaba	On the Bollobás – Eldridge conjecture for bipartite graphs
MPI-I-2003-1-008	P. Sanders	Polynomial Time Algorithms for Network Information Flow
MPI-I-2003-1-007	H. Tamaki	Alternating cycles contribution: a strategy of tour-merging for the traveling salesman problem
MPI-I-2003-1-006	M. Dietzfelbinger, H. Tamaki	On the probability of rendezvous in graphs
MPI-I-2003-1-005	M. Dietzfelbinger, P. Woelfel	Almost Random Graphs with Simple Hash Functions
MPI-I-2003-1-004	E. Althaus, T. Polzin, S.V. Daneshmand	Improving Linear Programming Approaches for the Steiner Tree Problem
MPI-I-2003-1-003	R. Beier, B. Vöcking	Random Knapsack in Expected Polynomial Time
MPI-I-2003-1-002	P. Krysta, P. Sanders, B. Vöcking	Scheduling and Traffic Allocation for Tasks with Bounded Splittability
MPI-I-2003-1-001	P. Sanders, R. Dementiev	Asynchronous Parallel Disk Sorting
MPI-I-2002-4-002	F. Drago, W. Martens, K. Myszkowski, H. Seidel	Perceptual Evaluation of Tone Mapping Operators with Regard to Similarity and Preference
MPI-I-2002-4-001	M. Goesele, J. Kautz, J. Lang, H.P.A. Lensch, H. Seidel	Tutorial Notes ACM SM 02 A Framework for the Acquisition, Processing and Interactive Display of High Quality 3D Models
MPI-I-2002-2-008	W. Charatonik, J. Talbot	Atomic Set Constraints with Projection
MPI-I-2002-2-007	W. Charatonik, H. Ganzinger	Symposium on the Effectiveness of Logic in Computer Science in Honour of Moshe Vardi
MPI-I-2002-1-008	P. Sanders, J.L. Träff	The Factor Algorithm for All-to-all Communication on Clusters of SMP Nodes
MPI-I-2002-1-005	M. Hoefer	Performance of heuristic and approximation algorithms for the uncapacitated facility location problem
MPI-I-2002-1-004	S. Hert, T. Polzin, L. Kettner, G. Schäfer	Exp Lab A Tool Set for Computational Experiments
MPI-I-2002-1-003	I. Katriel, P. Sanders, J.L. Träff	A Practical Minimum Scanning Tree Algorithm Using the Cycle Property
MPI-I-2002-1-002	F. Grandoni	Incrementally maintaining the number of l-cliques
MPI-I-2002-1-001	T. Polzin, S. Vahdati	Using (sub)graphs of small width for solving the Steiner problem
MPI-I-2001-4-005	H.P.A. Lensch, M. Goesele, H. Seidel	A Framework for the Acquisition, Processing and Interactive Display of High Quality 3D Models
MPI-I-2001-4-004	S.W. Choi, H. Seidel	Linear One-sided Stability of MAT for Weakly Injective Domain
MPI-I-2001-4-003	K. Daubert, W. Heidrich, J. Kautz, J. Dischler, H. Seidel	Efficient Light Transport Using Precomputed Visibility
MPI-I-2001-4-002	H.P.A. Lensch, J. Kautz, M. Goesele, H. Seidel	A Framework for the Acquisition, Processing, Transmission, and Interactive Display of High Quality 3D Models on the Web

MPI-I-2001-4-001	H.P.A. Lensch, J. Kautz, M. Goesele, W. Heidrich, H. Seidel	Image-Based Reconstruction of Spatially Varying Materials
MPI-I-2001-2-006	H. Nivelle, S. Schulz	Proceeding of the Second International Workshop of the Implementation of Logics
MPI-I-2001-2-005	V. Sofronie-Stokkermans	Resolution-based decision procedures for the universal theory of some classes of distributive lattices with operators
MPI-I-2001-2-004	H. de Nivelle	Translation of Resolution Proofs into Higher Order Natural Deduction using Type Theory
MPI-I-2001-2-003	S. Vorobyov	Experiments with Iterative Improvement Algorithms on Completely Unimodel Hypercubes