Online Topological Ordering

Irit Katriel

Hans L. Bodlaender

institute of information and computing sciences, utrecht university

technical report UU-CS-2005-011

www.cs.uu.nl

Online Topological Ordering

Irit Katriel[1] and Hans L. Bodlaender[2]

[1] Max-Planck-Institut für Informatik, Saarbrücken, Germany
irit@mpi-sb.mpg.de
[2] Institute of Information and Computing Sciences, Utrecht University, the Netherlands
hansb@cs.uu.nl

Abstract. It is shown that the problem of maintaining the topological order of the nodes of a directed acyclic graph while inserting m edges can be solved in $O(\min\{m^{3/2}\log n, m^{3/2} + n^2 \log n\})$ time, an improvement over the best known result of $O(mn)$. In addition, we analyze the complexity of the same algorithm with respect to the treewidth k of the underlying undirected graph. We show that the algorithm runs in time $O(mk \log^2 n)$ for general k and that it can be implemented to run in $O(n \log n)$ time on trees, which is optimal. If the input contains cycles, the algorithm detects this.

1 Introduction

In this paper, we study online algorithms to maintain a topological ordering of a directed acyclic graph. In a topological ordering, each node $v \in V$ of a given directed acyclic graph (DAG) $G = (V, E)$ is associated with a value $ord(v)$, such that for each directed edge $(u, v) \in E$, we have $ord(u) < ord(v)$ [15]. In the online variant of the topological ordering 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 DAG, we are required to update the mapping ord. One directly observes that there are two cases when an edge is added. Suppose we have a DAG and a valid topological ordering function ord, and that an edge (x, y) is added to this DAG. If $ord(x) < ord(y)$ then ord is a valid topological ordering function for the new DAG and no updating is necessary. Otherwise, the new edge (x, y) is said to *violate* the topological order ord. In this case, we need to find a new function ord' which is a valid topological ordering function for the new DAG.

The online topological ordering problem has several applications. It has been studied in the context of compilation [7, 10] where dependencies between modules are maintained to reduce the amount of recompilation performed when an update occurs, source code analysis [11] 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]. Dynamic topological ordering is also used at the core of constraint systems for stochastic search such as those featured in Comet [9].

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. [8] gave an algorithm that can insert m edges in $O(mn)$ time, giving an amortized time bound of $O(n)$ per edge instead of the trivial $O(m)$. This is the best amortized result known so far. Pearce and Kelly [11] propose a different algorithm and show experimentally that it achieves a speedup on sparse inputs, although its worst case running time is slightly higher. It was shown [6] that the algorithm by Pearce and Kelly is worst-case optimal in terms of the number of node relabelling operations it performs.

[1] Here and in the rest of the paper we assume $m = \Omega(n)$.

Alpern et al. [1] designed an algorithm which runs in time $O(\|\delta\| \log \|\delta\|)$ in the *bounded incremental computation* model [13]. In this model, the parameter $\|\delta\|$ measures, upon an edge insertion, the size (number of nodes and edges) of a minimal subgraph that we need to update in order to obtain a valid topological order ord' for the modified graph. Since $\|\delta\|$ can be anywhere between 0 and $\Theta(m)$, 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.

The only non-trivial lower bound for online topological ordering is due to Ramalingam and Rep [12], who show that an adversary can force any algorithm to perform $\Omega(n \log n)$ node relabelling operations while inserting $n-1$ edges (and creating a chain). Thus, if the labels are maintained explicitely, this implies an $\Omega(n \log n)$ bound on runtime.

1.2 Our Results.

In Section 2 we show that a slight variation on the algorithm by Alpern et al. can handle m edge insertions in $O(\min\{m^{3/2} \log n, m^{3/2} + n^2 \log n\})$ time. This implies an amortized time of $O(\min\{\sqrt{m} \log n, \sqrt{m} + \frac{n^2 \log n}{m}\})$ per edge; an improvement over the previous result. We then show that this analysis is almost tight. That is, for any $m \leq n(n-1)/2$, there is an input of m edges which will take the algorithm $\Omega(m^{3/2})$ time to process.

For $m = n-1$ our upper bound is $O(n^{3/2} \log n)$, which is far from the lower bound of $\Omega(n \log n)$. However, the worst case input, on which the algorithm performs $\Omega(n^{3/2})$ work, contains bipartite cliques. In Section 3 we analyze the complexity of the same algorithm on structured graphs, and show that it has an implementation which is optimal on trees (assuming that the running time is at least proportional to the number of relabelling operations performed) and in general runs in time $O(mk \log^2 n)$, where k is the treewidth of the input DAG. While much of the current results that exploit the treewidth of graphs show that problems can be solved faster when the treewidth of the input graph is bounded, it is interesting to note that the notion also has other uses. For our result, we do not need to assume some bound on the treewidth, and we analyze an algorithm that does not use the treewidth or a tree decomposition of the given graph. Also, little research has so far been done on the effect of treewidth on online algorithms.

2 An $O(\min\{\sqrt{m} \log n, \sqrt{m} + \frac{n^2 \log n}{m}\})$ Amortized Upper Bound

Figure 5 shows a slight variation on the algorithm by Alpern et al. [1]. This variation works as follows[2]. The *ord* labels of the nodes are maintained by an *Ordered List* data structure *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 [2,4]: *InsertAfter*(x, y) (*InsertBefore*(x, y)) inserts the item x immediately after (before) the item y in the total order, *Delete*(x) removes the item x, the query *Order*(x, y) determines whether x precedes y or y precedes x in the total order and *Next*(x) (*Prev*(x)) returns the item that appears immediately after (before) x in the total order. These operations are implemented by associating integer labels with the items in the list such that the label associated with x is smaller than the label associated with y iff x precedes y in the total order.

Initially, no edges exist in the graph so the nodes are inserted into *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 *AddEdge* works. 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 of *AR* which are reachable from *Target* by a path in the DAG and the set *ToSource* is the set of nodes in *AR*

[2] The proof of correctness of the algorithm appears in [1]; here we are only describing how it operates and providing some intuition.

from which *Source* can be reached by a path. In the example in Figure 1, the nodes appear by the order of their *ord* labels and the dashed edge (I, C) is inserted. There, $AR = \{C, D, E, F, G, H, I\}$, $\mathit{FromTarget} = \{C, D, F\}$ and $\mathit{ToSource} = \{E, H, I\}$.

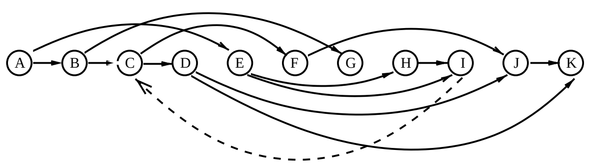

Fig. 1. The dashed edge violates the order *ord*.

Clearly, in any topological order for the modified graph, all nodes in *FromTarget* appear after all nodes in *ToSource*. The relative order among other nodes can remain unchanged, as well as the relative order within *FromTarget* and within *ToSource*. As illustrated in Figure 2, this can be achieved by relabelling the nodes of a subset of $\mathit{FromTarget} \cup \mathit{ToSource}$: Place a vertical line at an arbitrary location between *Target* and *Source*. Let *FromT* be the subset of *FromTarget* which is to the left of the line and *ToS* the subset of *ToSource* which is to the right of the line. Now, place all nodes in *FromT* immediately after the line and all nodes in *ToS* immediately before the line, while preserving the previous relative order within each set.

Conceptually, the algorithm begins with two horizontal lines, one immediately to the left of *Target* and the other immediately to the right of *Source*. The two lines sweep towards each other until they meet, and as they advance, the left line inserts into *FromT* all nodes it sweeps over which are in *FromTarget* and the right line inserts into *ToS* all nodes it sweeps over which are in *ToSource*. It remains to determine at which speed each line progresses. For this, we define two sets of edges: E_{FromT} of the edges outgoing from nodes in *FromT* and E_{ToS} of edges incoming into nodes in *ToS*. The algorithm aims to balance the cardinalities of E_{FromT} and E_{ToS} as much as possible. It begins to explore the nodes in *FromTarget* and *ToSource*, the first by starting at *Target* and advancing by order of the *ord* labels of the nodes, and the second by starting at *Source* and advancing in reverse order of the *ord* labels. Figure 3 shows how the algorithm would select the location of the line in the graph of Figure 2.

In practice, this is implemented as follows. Two priority queues are used for the sweeping: *SourceQueue* for the nodes that can reach nodes in *ToS* and *TargetQueue* for the nodes that are reachable from nodes in *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.

For each node $v \in V$, let $\mathit{InDegree}[v]$ be the number of edges incoming into v and $\mathit{OutDegree}[v]$ be the number of edges outgoing from v. Once a node becomes a candidate for insertion into *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 $\mathit{InDegree}[v]$ ($\mathit{OutDegree}[v]$). 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. This means that for every edge outgoing from a node in *FromTarget*, there is an edge incoming into a node in $\mathit{ToSource} \cup \{s\}$, and vice versa. When a node is inserted into *ToS* (*FromT*), its incoming (outgoing) edges are traversed and each of its predecessors (successors) is inserted into *SourceQueue* (*TargetQueue*) if it is not already there. It is important to note that the time spent by the algorithm is proportional to the number of edges adjacent to nodes that were inserted into *ToS* and *FromT*, and does not depend on the degrees of the last candidates for insertion into these sets.

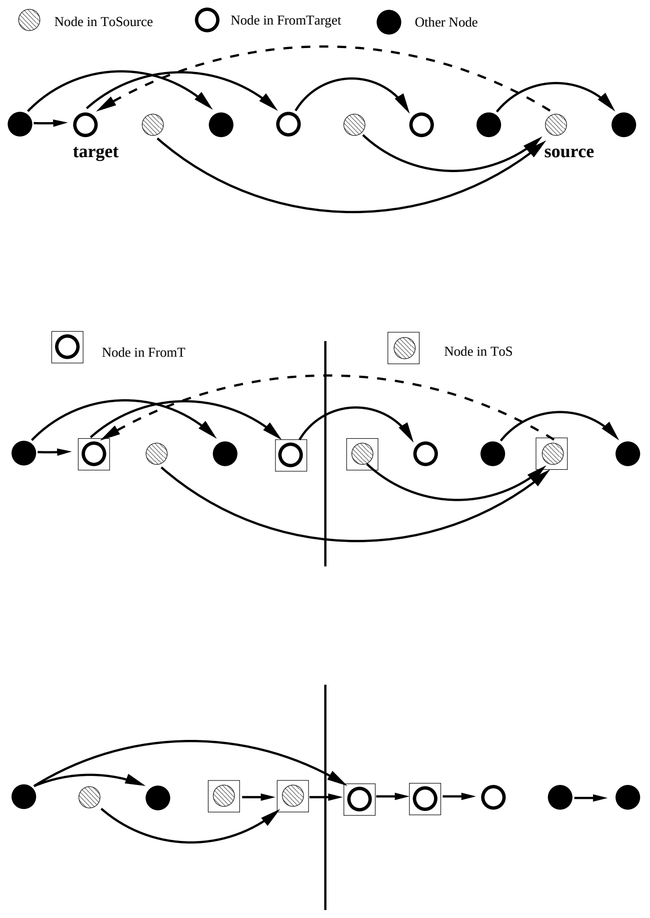

Fig. 2. Illustration of the intuition behind the algorithm. The top image shows an input graph, a new edge and the sets *ToSource* and *FromTarget*. The middle image shows the sets *ToS* and *FromT* with respect to the drawn line. The bottom image shows how to obtain a topological order for the modified graph by moving only the nodes of $ToS \cup FromT$.

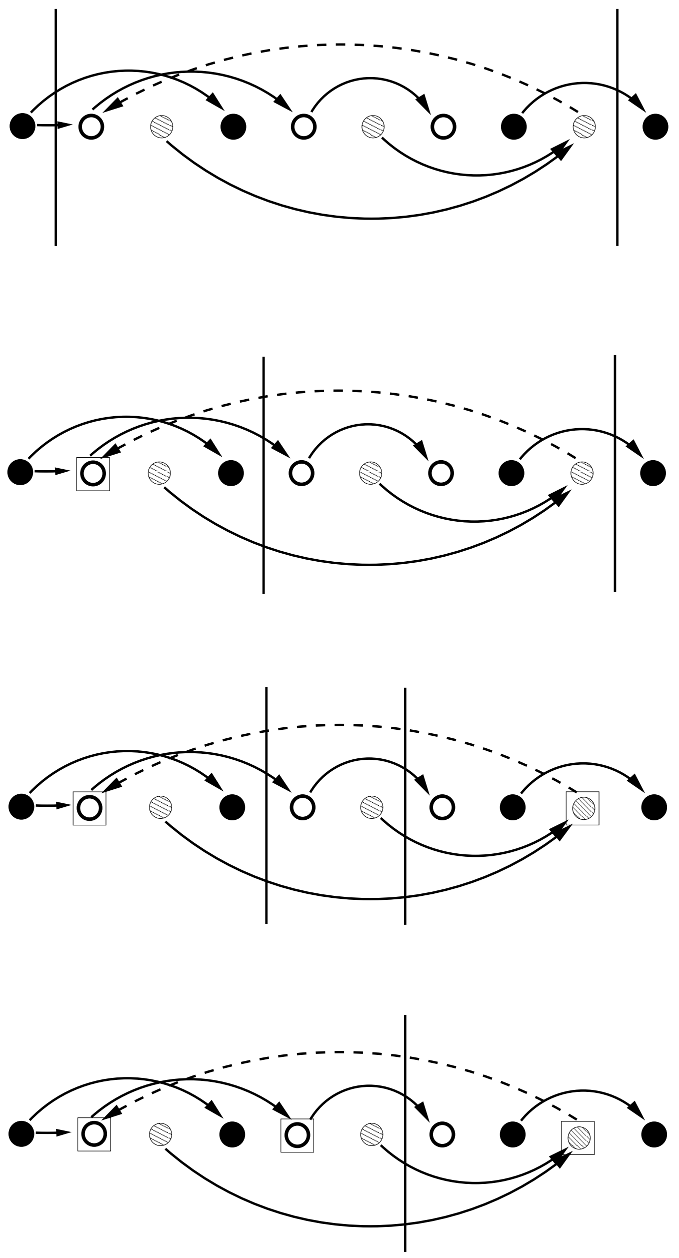

Fig. 3. Two lines sweeping towards each other until they merge. The left line constructs the set *FromT* and the right line constructs the set *ToS*.

The construction of the sets *ToS* and *FromT* ends when $ord(s) \leq ord(t)$ or $ToS = ToSource$ or $FromT = FromTarget$ or the algorithm discovers that the edge (*Source*, *Target*) introduces a cycle in the graph. In the later case, the algorithm terminates. Otherwise, the nodes in *ToS* are deleted from *ORD* and are reinserted, in the same relative order among themselves, before all nodes in $FromTarget \setminus FromT$ and the nodes in *FromT* are deleted and reinserted in the same relative order among themselves after all nodes in *ToSource*. Finally, the edge (*Source*, *Target*) is inserted into the DAG. In the example in Figure 1, $FromT = \{C, D\}$, $ToS = \{E, H, I\}$ and after relabelling the nodes appear in *ORD* in the order shown in Figure 4.

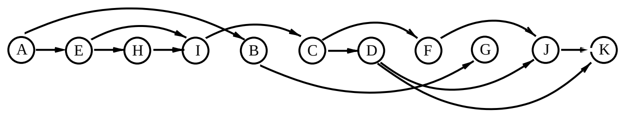

Fig. 4. The graph of Figure 1 after relabelling the nodes and inserting the edge (I, C).

Remark. We have mentioned above that the algorithm we describe is a variation on the one by Alpern et al. The first 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 incremental complexity model, their definition is appropriate because it is the sum of the total degrees that should be minimized. For our analysis, however, this change is necessary. In addition, we have simplified the relabelling phase, compared to what was proposed by Alpern et al. Their relabelling phase minimizes the number of different labels used. While this gives a worse asymptotic complexity, they claim that it speeds up the operations on the Ordered List data structure in practice.

Cycle Detection

For simplicity, we omitted cycle detection from Figure 5. This can be added as follows: First, if $Source = Target$ then the new edge forms a self-loop, i.e., a cycle. Otherwise, we maintain two flags for each node, an s-flag and a t-flag. Initially, both flags are set to 0. When a node is inserted into *SourceQueue* (*TargetQueue*), its s-flag (t-flag) is set to 1. The flag remains 1 even after the node is extracted from the queue. If there is a node with both flags equal to 1, the new edge introduces a cycle.

2.1 Complexity Analysis.

The complexity analysis consists of two parts. In the first part, we show that the time spent on m edge insertions is $O(m^{3/2} \log n)$. In the second part, we show that this time is $O(m^{3/2} + n^2 \log n)$. The combination of both results gives:

Theorem 1. *The topological order of the nodes of a DAG can be maintained while inserting m edges in total time $O(\min\{m^{3/2} \log n, m^{3/2} + n^2 \log n\})$.*

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 there is a path from y_i to x_j or from y_j to x_i that uses only edges from $\{e_1, \ldots, e_k\}$. That is, if the edges that were already inserted to the DAG form a path that determines the relative positions of e_i and e_j in the topological order. Otherwise, we say that the edges are *unordered*. Similarly, we say that a pair of nodes u, v is ordered if there is a path from u to v or from v to u, and unordered otherwise.

We denote by $\mathcal{U}_e \subseteq E \times E$ the set of unordered pairs of edges and by $\mathcal{U}_v \subseteq V \times V$ the set of unordered pairs of nodes.

```
Function $AddEdge$ ($Source$ , $Target$ )
    $SourceQueue \leftarrow []$; $TargetQueue \leftarrow []$
    $SourceStack \leftarrow []$ ; $TargetStack \leftarrow []$
    $s \leftarrow Source$ ; $t \leftarrow Target$
    $SourceDegree \leftarrow InDegree[s]$
    $TargetDegree \leftarrow OutDegree[t]$

    (* Discovery: decide which nodes should be relabelled. *)
    while $(ord(s) > ord(t))$ do
        $d \leftarrow \min\{SourceDegree, TargetDegree\}$
        $SourceDegree \leftarrow SourceDegree - d$
        $TargetDegree \leftarrow TargetDegree - d$
        if $SourceDegree = 0$ then
            foreach $(w, s) \in E$ do $SourceQueue.Insert(w)$
            $SourceStack.Push(s)$
            if $SourceQueue.NotEmpty$ then
                $s \leftarrow SourceQueue.ExtractMax$
            else $s \leftarrow ORD.Prev(Target)$ (* Then we are done *)
            $SourceDegree \leftarrow InDegree[s]$
        endif
        if $TargetDegree = 0$ then
            foreach $(t, w) \in E$ do $TargetQueue.Insert(w)$
            $TargetStack.Push(t)$
            if $TargetQueue.NotEmpty$ then
                $t \leftarrow TargetQueue.ExtractMin$
            else $t \leftarrow ORD.Next(Source)$ (* Then we are done *)
            $TargetDegree \leftarrow OutDegree[t]$
        endif
    end while

    (* Relabel the nodes *)
    if $s = Target$ then $s \leftarrow ORD.Prev(Target)$
    while $SourceStack.NotEmpty$ do
        $s' \leftarrow SourceStack.Pop$
        $ORD.Delete(s')$; $ORD.InsertAfter(s', s)$; $s \leftarrow s'$
    end while
    if $t = Source$ then $t \leftarrow ORD.Next(Source)$
    while $TargetStack.NotEmpty$ do
        $t' \leftarrow TargetStack.Pop$
        $ORD.Delete(t')$; $ORD.InsertBefore(t', t)$; $t \leftarrow t'$
    end while

    (* Insert the edge *)
    $E \leftarrow E \cup \{(Source, Target)\}$
    $InDegree[Target] \leftarrow InDegree[Target] + 1$
    $OutDegree[Source] \leftarrow OutDegree[Source] + 1$
end
```

Fig. 5. A variation on the algorithm by Alpern et al.

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 $\ell_t = 0$ ($\ell_s = 0$).

2.2 An $O(m^{3/2} \log n)$ Upper Bound.

We define a potential function $\Phi = |\mathcal{U}_e|$. I.e., the potential is equal to the number of unordered pairs of edges. Initially, all pairs of edges are unordered, so $\Phi_0 = m(m-1) < m^2$.

Lemma 1. *The algorithm spends on an insertion $O(1 + \max\{m_t, m_s\} \log n)$ 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 $n_i \le m_i$ for $i \in \{s,t\}$, we have that the time spent is $O(1+m_t \log m_t + m_s \log m_s) = O(1+(m_t+m_s)\log m) = O(1 + \max\{m_t, m_s\} \log n)$. □

Lemma 2. *For every edge insertion, after every iteration of the while loop of the discovery phase, $m_t + \ell_t - \mathit{TargetDegree} = m_s + l_s - \mathit{SourceDegree}$.*

Proof. We show this by induction on the number of iterations of the while loop which constructs the sets *FromT* and *ToS*.

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 $\mathit{val} \in \{m_t, \ell_t, m_s, \ell_s, \mathit{TargetDegree}, \mathit{SourceDegree}\}$, let val denote this value after iteration $i-1$ and val' 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 $\mathit{SourceDegree} \le \mathit{TargetDegree}$. 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 into *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 $m'_s + \ell'_s - \mathit{SourceDegree}'$. □

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

Proof. Since at all times $0 \le \mathit{TargetDegree} \le \ell_t$ and $0 \le \mathit{SourceDegree} \le \ell_s$, (1) and (2) follow from Lemma 2. □

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

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

Proof. Let $e_t = (x_t, y_t)$ be one of the m_t edges outgoing from a node in *FromT* and $e_s = (x_s, y_s)$ be one of the m_s edges incoming into a node in *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 into their sets in the same iteration of the discovery phase when the algorithm processed the insertion of the edge $(\mathit{Source}, \mathit{Target})$, then their relative order in ORD would imply that the algorithm terminates without adding either one of them, a contradiction. Assume, w.l.o.g., that y_s was inserted before x_t became the candidate. Then when 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 should have terminated without adding x_t. We get that the potential decreased by at least $m_t m_s$.

If in the last iteration, a node was added to both ToS and FromT, then by Lemma 2, $m_s = m_t$ and our claim follows. Otherwise, assume, w.l.o.g., that in the last iteration of the discovery phase, ToS grew by a node and FromT did not. Then the argument above holds also when e_t is one of the ℓ_t edges outgoing from the last node that was a candidate for insertion into FromT. We get that the potential decreased by at least $(m_t + \ell_t)m_s$. By Corollary 1, $(m_t + \ell_t)m_s \geq m_s^2$. Let $m_s', m_t', \ell_s', \ell_t', \mathit{SourceDegree}', \mathit{TargetDegree}'$ be the values of these variables before the last iteration. Then $m_s = m_s' + \ell_s' = m_t' + \ell_t' - (\mathit{TargetDegree}' - \mathit{SourceDegree}') = m_t + \ell_t - (\mathit{TargetDegree}' - \mathit{SourceDegree}') \geq m_t$, so we also have $(m_t + \ell_t)m_s \geq m_t^2$. □

Theorem 2. *The algorithm in Figure 5 needs $O(m^{3/2} \log n)$ time to insert m edges into an initially empty n-node graph.*

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

For $i = 1, \ldots, m$, let x_i be the value of $\max\{m_t, m_s\}$ 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 $\Phi_m - \Phi_0 = \sum_{i=1}^m \Delta\Phi_i \leq -\sum_{i=1}^m x_i^2$. Since $\Phi_0 < m^2$ and $\Phi_m \geq 0$, we have $\Phi_m - \Phi_0 > -m^2$ which implies $\sum_{i=1}^m x_i^2 < m^2$.

Cauchy's inequality states that

$$\left(\sum_{i=1}^m a_i b_i\right)^2 \leq \left(\sum_{i=1}^m a_i^2\right)\left(\sum_{i=1}^m b_i^2\right).$$

By substituting $a_i = x_i$ and $b_i = 1$ for all $1 \leq i \leq m$, we get that $\sum_{i=1}^m x_i < m^{3/2}$, so the total time spent by the algorithm on the m edge insertions is $O(m + \sum_{i=1}^m x_i \log n) = O(m^{3/2} \log n)$. □

2.3 An $O(m^{3/2} + n^2 \log n)$ Upper Bound.

In this section we change two aspects of the analysis, compared to the previous section. The first is the potential function - it now counts not only unordered pairs of edges but also unordered pairs of nodes: $\Psi = |\mathcal{U}_e| + |\mathcal{U}_v|$.

The second change is in the analysis of the actual time spent by the algorithm when it processes the insertion of an edge. We assume that the priority queues are implemented by a data structure that supports insertions in constant amortized time and extractions in $O(\log n)$ amortized time (e.g., Fibonacci Heaps [5]). Note that the number of items inserted into $\mathit{SourceQueue}$ can be as large as m_s, but the number of items extracted from it is bounded by n_s. This implies that the time spent on identifying the set ToS during the discovery phase is $O(m_s + n_s \log n)$, which is a tighter bound than the one we had in the previous section, of $O(m_s \log n)$. Similarly, the time spent on identifying the set FromT is $O(m_t + n_t \log n)$. Thus, we have shown:

Lemma 4. *The algorithm spends $O(1 + \max\{m_t, m_s\} + \max\{n_t, n_s\} \log n)$ time on an insertion.*

We now analyze the change in potential due to an insertion.

Lemma 5. *When the algorithm handles an insertion, $\Delta\Psi_i \leq -\max\{m_t^2, m_s^2\} - \max\{n_t, n_s\}$. I.e., the potential decreases by at least* $\max\{m_t^2, m_s^2\} + \max\{n_t, n_s\}$.

Proof. We have shown in the proof of Lemma 3 that $|\mathcal{U}_e|$ decreases by at least $\max\{m_t^2, m_s^2\}$. In addition, for each $v \in \mathit{FromT}$ ($v \in \mathit{ToS}$), the pair $\langle \mathit{Source}, v\rangle$ ($\langle v, \mathit{Target}\rangle$) was ordered after the insertion. If it was also ordered before the insertion, then the new edge introduces a cycle in the graph, contradicting our assumption that it is a DAG. So $|\mathcal{U}_v|$ decreases by at least $|\mathit{FromT}| + |\mathit{ToS}| = n_t + n_s \geq \max\{n_t, n_s\}$. □

Theorem 3. *The algorithm in Figure 5 needs $O(m^{3/2} + n^2 \log n)$ time to insert m edges into an initially empty n-node DAG.*

Proof. By Lemma 4, an insertion takes $O(1 + \max\{m_t, m_s\} + \max\{n_t, n_s\} \log n)$ time while by Lemma 5 the potential decreases by at least $\max\{m_t^2, m_s^2\} + \max\{n_t, n_s\}$, with a decrease of at least $\max\{m_t^2, m_s^2\}$ in the term $|\mathcal{U}_e|$ and a decrease of at least $\max\{n_t, n_s\}$ in the term $|\mathcal{U}_v|$.

For $i = 1, \ldots, m$, let x_i be the value of $\max\{m_t, m_s\}$ and y_i be the value of $\max\{n_t, n_s\}$ upon the insertion of the ith edge. We get that the total time spent on the m insertions is $O(\sum_{i=1}^{m}(1 + x_i + y_i \log n)) = O(m + \sum_{i=1}^{m} x_i + \log n \sum_{i=1}^{m} y_i)$ where $\sum_{i=1}^{m} x_i^2 \leq m^2$ and $\sum_{i=1}^{m} y_i \leq n^2$. As in the proof of Theorem 2, $\sum_{i=1}^{m} x_i \leq m^{3/2}$ so the total time spent is $O(m + \sum_{i=1}^{m} x_i + \sum_{i=1}^{m} y_i \log n) = O(m^{3/2} + n^2 \log n)$. □

2.4 Almost Tightness of the Analysis.

We now show that our analysis is almost tight. That is, we show that for each $m \leq n(n-1)/2$ there is an input with n nodes and m edges that will take the algorithm in Figure 5 $\Omega(m^{3/2})$ time to process, which is equal to the upper bound for dense graphs and is merely a log factor away for sparse graphs.

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 constructs graph A in Figure 6 by inserting 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}). Graphs B,C and D of Figure 6 illustrate the first two insertions of this sequence. At the end of the second part, we end up with graph E of Figure 6.

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 *ORD*. 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* (possibly both) and will traverse at least k incident edges. Then, it will reverse the order of *Source* and *Target* in *ORD*. Thus, it will spend $\Omega(k)$ time for each edge insertion 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 The Complexity on Structured Graphs

The only non-trivial lower bound for online topological ordering is due to Ramalingam and Rep [12], who show that an adversary can force any algorithm to perform $\Omega(n \log n)$ node re-labelling operations while inserting $n-1$ edges (creating a chain). The upper bound we have shown for $m = n-1$ is $O(n^{3/2} \log n)$ time. We have shown that for any m we can construct an input on which the algorithm performs $\Omega(m^{3/2})$ work, but this input contains bipartite cliques.

In this section we show that the algorithm performs much better when the graph is structured. In particular, we show that for any DAG G, the algorithm runs in time $O(mk \log^2 n)$ where k is

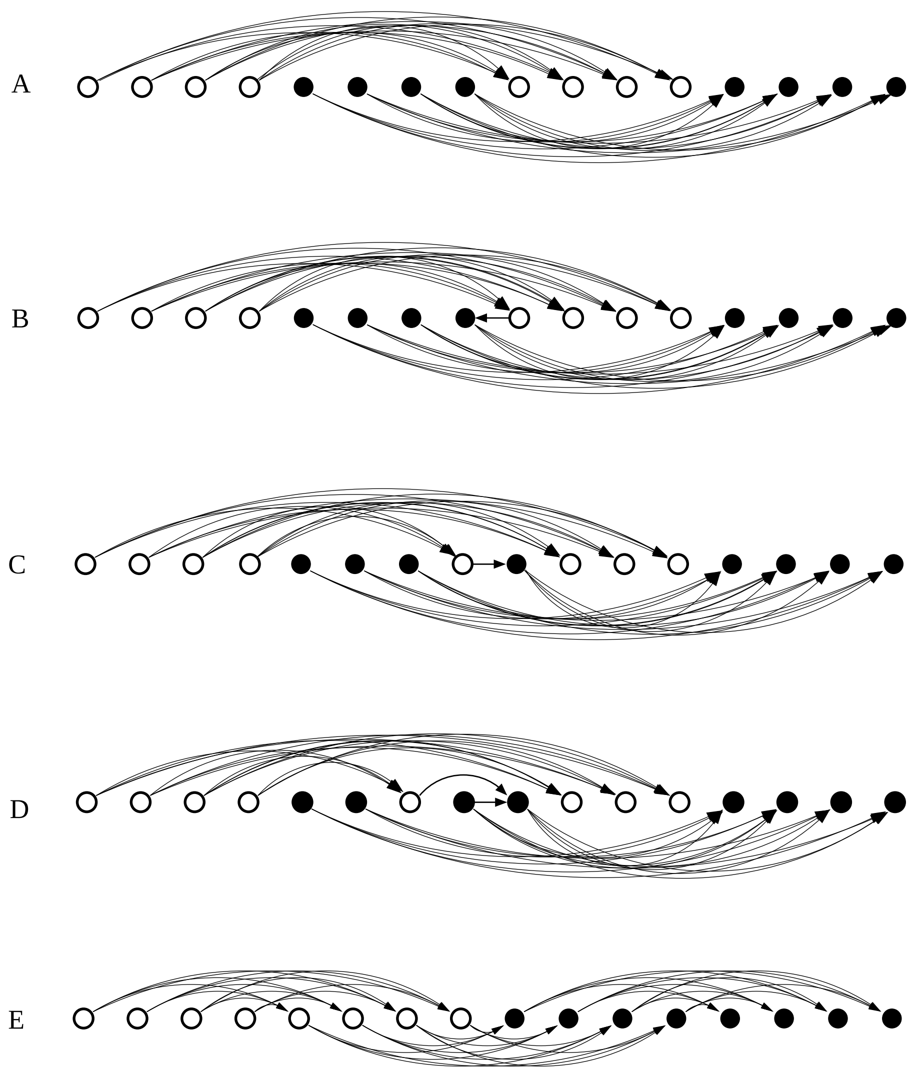

Fig. 6. An input that requires $\Omega(m^{3/2})$ time ($k = 4$).

the treewidth of G. In addition, we show that the algorithm can be implemented such that on a tree it spends a total of $O(n \log n)$ time, i.e., for trees, the algorithm is optimal and the lower bound is tight. The notion of treewidth was introduced by Robertson and Seymour [14]. We start with giving the definition of treewidth.

Definition 4 *A* tree decomposition *of an undirected graph $G = (V, E)$ is a pair $(\{X_i \mid i \in I\}, T = (I, F))$ with T a tree, and $\{X_i \mid i \in I\}$ a family of subsets of V, such that $\bigcup_{i \in I} X_i = V$, for all $\{v, w\} \in E$, there is an $i \in I$ with $v, w \in X_i$, and for all $v \in V$, the set of nodes $T_v = \{i \in I \mid v \in X_i\}$ induces a connected subgraph (subtree) of T.*

The width *of a tree decomposition $(\{X_i \mid i \in I\}, T)$ is $\max_{i \in I} |X_i| - 1$. The treewidth of a graph G is the minimum width of a tree decomposition of G. The treewidth of a directed graph G is the treewidth of the underlying undirected graph of G.*

Trees have treewidth one, while a bipartite clique with ℓ nodes on each side (as in Figure 6) has treewidth ℓ. For an overview of the treewidth of several classes of graphs, see e.g., [3].

Lemma 6. *The algorithm can be implemented such that if G is a tree, it performs a total of $O(n \log n)$ work.*

Proof. Let T_v be the tree that node v belongs to. When an edge (u, v) is inserted, two trees T_u and T_v are joined into one tree. Let $|T|$ be the number of nodes in tree T and assume, w.l.o.g., that $|T_u| \leq |T_v|$. Then the algorithm can be implemented such that it relabels $O(|T_u|)$ nodes and both the discovery and relabelling phases take $O(|T_u|)$ time. To do this, we need to give up the priority queues used in the discovery phase. The predecessors of *Source* (successors of *Target*) will be traversed in DFS order and will appear in the new topological order according to their relative postorder DFS numbers. Since we do not examine the nodes of *ToSource* (*FromTarget*) in increasing *ord* order, the discovery phase must continue until all of the nodes of either *ToSource* or *FromTarget* have been traversed. Since the two traversals are performed simultaneously, the time they take is proportional to the size of the smaller of T_u and T_v.

Define the potential function $f = \sum_{v \in V} \log |T_v|$. When an edge (u, v) is inserted and the algorithm performs $O(|T_u|)$ work, the potential increases by $(|T_u| + |T_v|) \log(|T_u| + |T_v|) - |T_u| \log |T_u| - |T_v| \log |T_v| \geq |T_u|$. The lemma follows because the potential is initially 0 and it is never larger than $n \log n$. $\square$

We now turn to the analysis of graphs of treewidth k. We need a simple lemma, which is a small variant of a well known result. (Compare, e.g., [14].) The proofs of the three lemmas below follow standard techniques.

Lemma 7. *Let T be a tree with ℓ leaves. Then there is a node v in T such that each connected component of $T - v$ contains at most $\ell/2$ leaves from T.*

Proof. Suppose the lemma does not hold for tree T. For each node v from T, there is then exactly one subtree of $T - v$ with more than $\ell/2$ leaves. Build a directed graph H, by taking for each node v a directed edge to its neighbor in this subtree. As each node in H has outdegree one, H must contain a cycle, hence two neighboring nodes v_0, v_1 with edges (v_0, v_1) and (v_1, v_0) in H. The subtree of $T - v_0$ that contains v_1 is disjoint from the subtree of $T - v_1$ that contains v_0, and both these subtrees contain more than $\ell/2$ leaves, so T has more than ℓ leaves, contradiction. $\square$

Lemma 8. *Let $G = (V, E)$ be a graph with treewidth k. There is a set $S \subseteq V$ such that $|S| \leq k+1$, and for each connected component of $G - S$ there are at most $m/2$ edges with at least one endpoint in the component.*

Proof. Take a tree decomposition $(\{X_i \mid i \in I\}, T = (I, F))$ of G of width at most k. For each edge $e = \{v, w\} \in E$, add one node i_e to the tree decomposition with $X_{i_e} = \{v, w\}$, and make i_e adjacent in T to an (old) node $i' \in I$ with $v, w \in X_{i'}$. Let $(\{X_i \mid i \in I'\}, T' = (I', F'))$ be the resulting tree decomposition. We have $I' = I \cup \{i_e \mid e \in E\}$. While T' has one or more leaf nodes that belong to

I, i.e., do not correspond to an edge, remove such leaf nodes. Let $(\{X_i \mid i \in I'\}, T'' = (I'', F''))$ be the resulting tree decomposition. (One can verify that this is a tree decomposition of G of width at most k.) There is a one-to-one correspondence between the edges of E and the leaves of T''.

Let i^0 be the node in T'' such that each subtree of $T'' - i_0$ contains at most $m/2$ nodes that were a leaf in T'', i.e., correspond to an edge in E. Set $S = X_{i_0}$. Clearly, $|S| \leq k+1$. Consider a connected component of $G - S$. By the properties of tree decomposition, there can be only one subtree of $T - i_0$ whose sets X_i, i in the subtree, can contain nodes from the component. In particular, for each edge $e = \{x, y\}$ with one endpoint in the component, we have that i_e belongs to that subtree. So, all edges with one endpoint in the component correspond to a node in the subtree that is a leaf in T''. So, the component has at most $m/2$ edges with at least one endpoint in it. □

Lemma 9. *Let $G = (V, E)$ be a graph with treewidth k. There is partition of V in three disjoint subsets A, B, and S, such that $|S| \leq k+1$, no node in A is adjacent to a node in B, the number of edges with at least one endpoint in A is at most $2m/3$, and the number of edges with at least one endpoint in B is at most $2m/3$.*

Proof. Let S be as in Lemma 8. If $G - S$ has a connected component with at least $m/3$ edges, then put the nodes of this connected component in A, and the nodes of the other connected component in B, and we are done. Otherwise, put the nodes of the components one by one in A, until the components in A together have at least $m/3$ edges. Put the nodes of the remaining components in B. Now, there are at least $m/3$ and at most $2m/3$ edges with at least one endpoint in A, and hence also at most $2m/3$ edges with at least one endpoint in B. □

Except in some degenerate cases, the set S from Lemma 9 is a separator. Let S be the set as in Lemma 9. Now define the potential function

$$\Psi = \sum_{v \in S} \sum_{e \in E} I(v, e)$$

where $I(v, e)$ is 1 if v and e are ordered and 0 otherwise. That is, the potential function counts the number of ordered pairs of an edge from the graph and a node from S. Clearly, $\Psi_0 = 0$ and at all times $\Psi \leq |S||E| \leq (k+1)m$.

The edges can be partitioned into three sets: the sets E_A (E_B) of edges whose insertions cause only nodes from A (B) to be relabelled, and the set $E_M = E \setminus (E_A \cup E_B)$. Let $T(m)$ be the time spent during all m edge insertions. Then the insertion of all edges from E_A (E_B) occur within the subgraph induced by the nodes in A (B). The time spent on the insertion of the edges in E_A (E_B) can be bounded by $T(|E{\downarrow}A|)$ ($T(|E{\downarrow}B|)$), where $E{\downarrow}A$ ($E{\downarrow}B$) is the number of edges that have at least one endpoint in A (B). To see why this holds, note that it is true even if $E{\downarrow}A = E_A$ ($E{\downarrow}B = E_B$). The fact that some of the edges in $E{\downarrow}A \cup E{\downarrow}B$ are in E_M, only reduces the complexity of inserting $E_A \cup E_B$.

While handling the insertion of an edge $e \in E_M$, the algorithm relabels at least one node v from S. Assume, w.l.o.g., that $v \in$ *ToSource*. Then by considerations similar to the ones used in the proof of Lemma 3, there are $(m_t + \ell_t)$ edges outgoing from nodes in *FromTarget* that were not ordered with v before the insertion of e and are ordered afterwards. Hence, $\Delta\Psi \geq m_t + \ell_t$, which by Corollary 1 is at least $\max\{m_s, m_t\}$. Since the time spent is $O(\max\{m_s, m_t\} \log n)$, we obtain that the insertion of all edges from E_M requires a total of $O(mk \log n)$ time. Hence, we have for the total time that $T(m) \leq mk \log n + T(|E_A|) + T(|E_B|)$, and as $|E_A| + |E_B| \leq m$, and $|E_A| \leq 2m/3$, $|E_B| \leq 2m/3$, it follows that $T(m) = O(mk \log^2 n)$. So we have shown:

Theorem 5. *The topological order of the nodes of a DAG with treewidth k can be maintained while inserting m edges into an initially empty n-node DAG in total time $O(mk \log^2 n)$. For the special case of trees, this problem can be solved in $O(n \log n)$ time, which is optimal.*

4 Conclusion

We have shown that the online topological ordering problem can be solved in $O(\min\{\sqrt{m}\log n, \sqrt{m}+\frac{n^2 \log n}{m}\})$ amortized time per edge, an improvement over the previous result of $O(n)$ time per edge. We have also shown that for any $m \leq n(n-1)/2$ there is an input with m edges on which the algorithm performs $\Omega(m^{3/2})$ work, indicating that our analysis of the algorithm's running time is almost tight if only the number of nodes and number of edges is known.

We then analyze the algorithm's complexity on structured graphs. We show that the algorithm has an optimal implementation on trees and that in general there is a correlation between the treewidth of the graph and the algorithm's complexity. There is here still a gap between upper and lower bound. Observe that for trees ($k = 1$), the general bound gives $O(n \log^2 n)$ and not the $O(n \log n)$ result that we have obtained separately. It is an open problem whether a different analysis or algorithm can yield an $O(mk \log n)$ upper bound.

Acknowledgements.

A special thanks is due to Alex Fabrikant for his excellent remarks on a previous version of this paper. We also wish to thank David J. Pearce, Seth Pettie, and Sven Thiel for helpful discussions. Part of this work has been done at the Dagstuhl seminar *Robust and Approximative Algorithms on Particular Graph Classes*.

References

1. B. Alpern, R. Hoover, B. K. Rosen, P. F. Sweeney, and F. Kenneth Zadeck. Incremental evaluation of computational circuits. In *Proc. of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 32–42. SIAM, 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 *Proc. of the 10th Annual European Symposium on Algorithms*, pages 152–164. Springer-Verlag, 2002.
3. Hans L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. *Theor. Comp. Sc.*, 209:1–45, 1998.
4. P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *Proc. of the 19th Annual ACM Conference on Theory of Computing*, pages 365–372. ACM Press, 1987.
5. M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. Assoc. Comput. Mach.*, 34:596–615, 1987.
6. I. Katriel. On algorithms for online topological ordering and sorting. Research Report MPI-I-2004-1-003, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 2004.
7. A. Marchetti-Spaccamela, U. Nanni, and H. Rohnert. On-line graph algorithms for incremental compilation. In *Proc. of International Workshop on Graph-Theoretic Concepts in Computer Science (WG 93)*, volume 790 of *LNCS*, pages 70–86, 1993.
8. A. Marchetti-Spaccamela, U. Nanni, and H. Rohnert. Maintaining a topological order under edge insertions. *Inf. Process. Lett.*, 59(1):53–58, 1996.
9. L. Michel and P. Van Hentenryck. A Constraint-Based Architecture for Local Search. In *Conference on Object-Oriented Programming Systems, Languages, and Applications.*, pages 101–110, Seattle, WA, USA, November 4-8 2002. ACM.
10. 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.
11. D. J. Pearce and P. H. J. Kelly. A dynamic algorithm for topologically sorting directed acyclic graphs. In *Proc. of the 3rd Int. Worksh. Efficient and Experimental Algorithms (WEA 2004)*, Lecture Notes in Computer Science. Springer-Verlag, 2004.
12. G. Ramalingam and T. Reps. On competitive on-line algorithms for the dynamic priority-ordering problem. *Inf. Process. Lett.*, 51(3):155–161, 1994.
13. G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *Theor. Comput. Sci.*, 158(1-2):233–277, 1996.
14. Neil Robertson and Paul D. Seymour. Graph minors. II. Algorithmic aspects of tree-width. *J. Algorithms*, 7:309–322, 1986.
15. R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, June 1972.