Engineering a Topological Sorting Algorithm for Massive Graphs

Deepak Ajwani* Adan Cosgaya-Lozano† Norbert Zeh‡

Abstract

We present an I/O-efficient algorithm for topologically sorting directed acyclic graphs (DAGs). No provably I/O-efficient algorithm for this problem is known. Similarly, the performance of our algorithm, which we call IterTS, may be poor in the worst case. However, our experiments show that IterTS achieves good performance in practise.

The strategy of IterTS can be summarized as follows. We call an edge satisfied if its tail has a smaller number than its head. A numbering satisfying at least half the edges in the DAG is easy to find: a random numbering is expected to have this property. IterTS starts with such a numbering and then iteratively corrects the numbering to satisfy more and more edges until all edges are satisfied.

To evaluate IterTS, we compared its running time to those of three competitors: PeelTS, an I/O-efficient implementation of the standard strategy of iteratively removing sources and sinks; ReachTS, an I/O-efficient implementation of a recent parallel divide-and-conquer algorithm based on reachability queries; and SeTS, standard DFS-based topological sorting built on top of a semi-external DFS algorithm. In our evaluation on various types of input graphs, IterTS consistently outperformed PeelTS and ReachTS, by at least an order of magnitude in most cases. SeTS outperformed IterTS on most graphs whose vertex sets fit in memory. However, IterTS often came close to the running time of SeTS on these inputs and, more importantly, SeTS was not able to process graphs whose vertex sets were beyond the size of main memory, while IterTS was able to process such inputs efficiently.

*MADALGO Center for Massive Data Algorithmics, Department of Computer Science, Aarhus University, Aarhus, Denmark. Email: ajwani@madalgo.au.dk. Research was supported in part by the Danish National Research Foundation. Travel to the conference was supported by IRCSET/IBM.

†Faculty of Computer Science, Dalhousie University, Halifax, NS, Canada. Email: acosgaya@cs.dal.ca. Supported in part by NSERC.

‡Faculty of Computer Science, Dalhousie University, Halifax, NS, Canada. Email: nzeh@cs.dal.ca. This research was supported in part by NSERC and the Canada Research Chairs programme.

1 Introduction

Let $G = (V, E)$ be a directed acyclic graph (DAG) with $n := |V|$ vertices and $m := |E|$ edges. *Topological sorting* is the problem of finding a linear ordering of the vertices in V such that the tail of each edge in E precedes its head in the ordering. Linear-time algorithms for this problem are covered in standard undergraduate texts, as topological sorting captures the problem of finding a linear order of items or activities consistent with a set of pairwise ordering constraints, which arises in a number of applications. The problem of topologically sorting large DAGs arises, for example, in the application of recent multiple sequence alignment algorithms [20,21] to large collections of DNA sequences.

Topologically sorting large DAGs is also an important building block for other I/O-efficient algorithms, mostly due to a technique called *time-forward processing* [9], which has proven useful in obtaining I/O-efficient solutions to a number of problems but requires the vertices of the graph to be given in topologically sorted order. Time-forward processing solves the following "graph evaluation" problem: given a DAG each of whose vertices has a label $\phi(x)$, process its vertices in topologically sorted order and, for each vertex x, compute a new label $\psi(x)$ from $\phi(x)$ and the ψ-labels of x's in-neighbours. A simple example of this type of problem is the evaluation of a Boolean circuit represented as a DAG: $\phi(\cdot)$ assigns a Boolean function to each vertex, turning it into a logical gate; $\psi(x)$ is the output of the gate represented by vertex x, given the inputs it receives from its in-neighbours. Since time-forward processing requires the vertices of the DAG to be given in topologically sorted order and no general I/O-efficient topological sorting algorithm is known to date, time-forward processing has been applied only in situations where a topological ordering of the vertices can be obtained by using secondary information about the structure of the DAG (e.g., [3, 4, 13, 15]). A general topological sorting algorithm for massive graphs would greatly increase the applicability of this technique.

Two simple linear-time algorithms for topological sorting are to repeatedly number and remove sources (in-degree-0 vertices) or to perform a depth-first search (DFS) of the graph and number the vertices in reverse

postorder [10]. Both approaches access the vertices in an unpredictable fashion and, thus, usually perform one random disk access per vertex when processing inputs beyond the size of main memory, while data access patterns with a high degree of locality are the key to I/O efficiency, as they facilitate the transfer of data between memory and disk in large blocks. This lack of locality in their data access patterns is a problem for all graph exploration strategies, not only for DFS, at least on directed graphs. Due to the strong reliance on such graph exploration strategies in traditional graph algorithms, even simple problems, such as topological sorting, become challenging on massive graphs.

A significant amount of work has focused on developing I/O-efficient graph algorithms. These algorithms are designed and analyzed in the *I/O model* [1], which assumes the computer is equipped with a two-level memory hierarchy consisting of *internal memory* and (disk-based) *external memory*. All computation has to happen in internal memory, which is capable of holding M data items. The transfer of data between internal and external memory happens by means of *I/O operations* (I/Os), each of which transfers a block of B consecutive data items to or from disk. The cost of an algorithm in this model is the number of I/Os it performs to solve the given problem.

For many problems on *undirected* graphs, the inefficiency of graph exploration techniques in the I/O model has been overcome by developing alternate techniques for solving graph problems that do not rely on graph exploration. For special graph classes, a wide range of problems, including topological sorting [5, 6, 14, 16], can be solved I/O-efficiently using techniques that exploit the structure of these graphs (e.g., planar separators). For *general directed* graphs, on the other hand, almost no I/O-efficient solutions to even the most elementary graph problems are known. The currently best general directed DFS and BFS (breadth-first search) algorithms perform $O((n + m/B) \log n)$ I/Os [8], which is efficient for dense graphs but worse than standard internal-memory DFS and BFS for sparse graphs. No techniques for solving problems on directed graphs without graph exploration are known, as even time-forward processing can be seen as exploring the graph from the sources toward the sinks.

The lack of *provably* I/O-efficient algorithms for directed graphs has led to the development of a number of heuristic approaches to solving problems on directed graphs I/O-efficiently. Most notably, Sibeyn et al. [18] proposed a DFS heuristic that performs extremely well if the vertex set of the graph fits in memory but breaks down on larger graphs. In [11], a contraction-based heuristic for computing the strongly connected components of a directed graph was proposed. In this paper, we propose an algorithm for topologically sorting directed acyclic graphs that falls into this category of efficient heuristics. In the worst case, its performance is poor, but our experiments show that it performs well in practise and can efficiently process graphs beyond the reach of existing algorithms, including an algorithm based on the DFS heuristic of [18].

The rest of this paper is organized as follows. In Section 2, we describe our new algorithm. In Section 3, we describe three algorithms we considered reasonable competitors. We implemented these algorithms and compared their performance with that of our algorithm. In Section 4, we present some implementation details and discuss our experimental setup and results. In Section 5, we give some concluding remarks.

2 Topological Sorting by Iterative Improvement (IterTS)

Our new topological sorting algorithm, called IterTS throughout this paper, is based on the following strategy. Given a numbering $\nu(\cdot)$ of the vertices of the DAG, we call an edge *satisfied* if its tail receives a lower number than its head; otherwise the edge is *violated*. The *satisfied subgraph* of the DAG G is a DAG G_ν whose vertex set is V and whose edge set consists of all edges of G satisfied by $\nu(\cdot)$. After computing an initial numbering $\nu_0(\cdot)$ and its corresponding satisfied subgraph G_{ν_0}, we proceed in iterations, each of which computes a new numbering $\nu_i(\cdot)$ from the previous numbering $\nu_{i-1}(\cdot)$, with the goal of increasing the number of satisfied edges. The computation of $\nu_i(\cdot)$ from $\nu_{i-1}(\cdot)$ ensures that $\nu_i(\cdot)$ satisfies strictly more edges than $\nu_{i-1}(\cdot)$. Thus, the algorithm is guaranteed to terminate, slowly in the worst case, quickly in practise.

Our description of the algorithm consists of four parts. In Section 2.1, we describe how we compute the initial numbering $\nu_0(\cdot)$. In Section 2.2, we discuss the computation in each iteration. In Section 2.3, we analyze the I/O complexity of the algorithm. In Section 2.4, we discuss a heuristic that led to a tremendous performance improvement.

2.1 Computing the Initial Numbering.

Throughout the algorithm, we assume G has only one source s. If this is not the case, we add a new source and connect it to each of the original sources. We compute an out-tree of s, that is, a spanning tree T_0 of G whose root is s and all of whose edges are directed away from s; see Figure 1(a). Since G is acyclic, such a spanning tree can be obtained by choosing, for each vertex $x \neq s$, an arbitrary in-edge to be included in T_0.

After choosing an arbitrary left-to-right ordering of

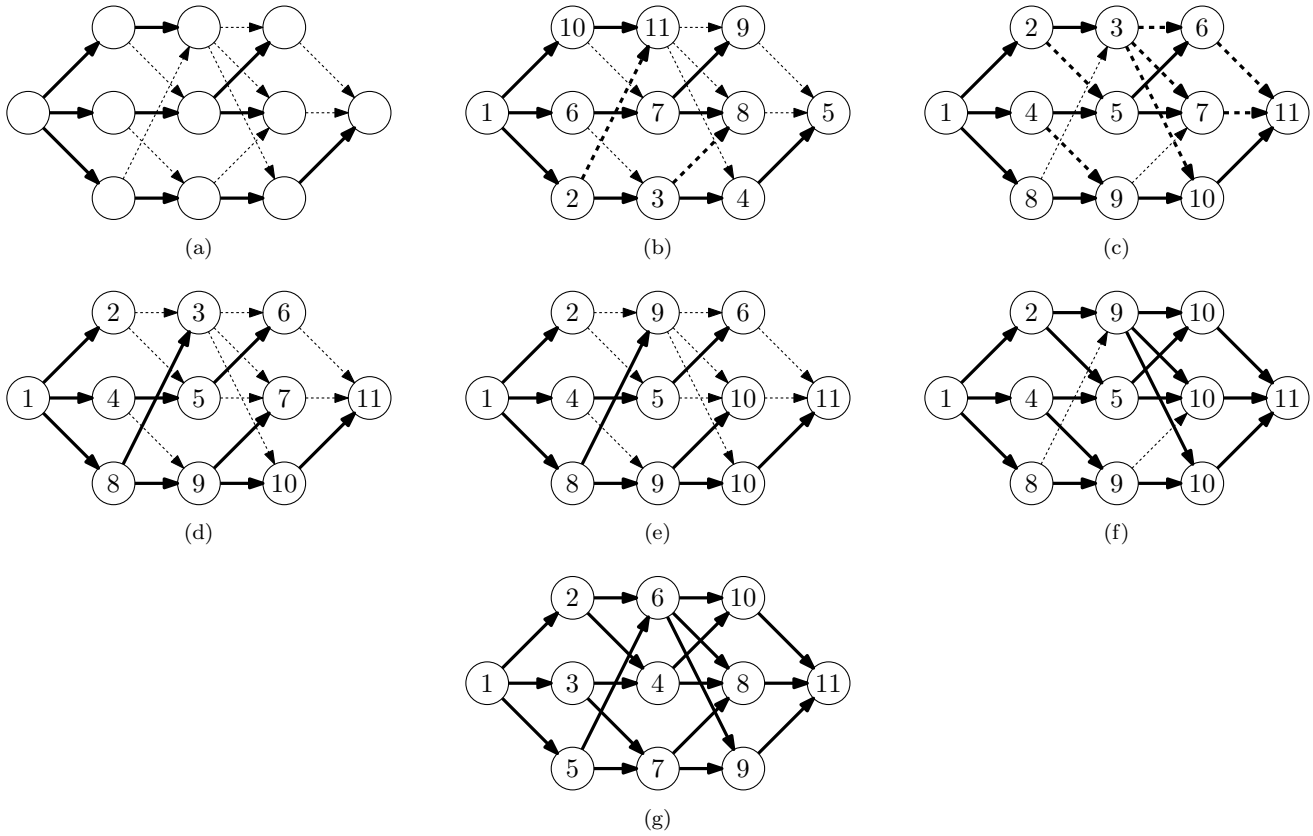


Figure 1: (a) A DAG with an out-tree T_0, shown in bold. (b/c) The two preorder numberings of T_0. The one in (c) satisfies more non-tree edges (bold dashed) and is the one we choose as the first numbering $\nu_0(\cdot)$. (d) The out-tree T_1 chosen in the first iteration. (e) The numbering $\nu'_1(\cdot)$ computed to satisfy all edges in T_1. (f) The numbering $\nu''_1(\cdot)$ computed to satisfy all edges in G_{ν_0}, shown in bold. (g) The numbering $\nu_1(\cdot)$ obtained by sorting the vertices according to $\nu''_1(\cdot)$ and then numbering them in order. The satisfied subgraph G_{ν_1}, shown in bold, contains all edges of G. Thus, $\nu_1(\cdot)$ is a topological ordering of G, and the algorithm terminates.

the out-edges of each vertex in T_0, we compute two numberings $\nu_l(\cdot)$ and $\nu_r(\cdot)$ of the vertices of T_0; see Figures 1(b) and 1(c). Both are preorder numberings of T_0; $\nu_l(\cdot)$ numbers the subtrees of each vertex in left-to-right order, while $\nu_r(\cdot)$ numbers the subtrees in right-to-left order. It is easy to see that one of these two numberings satisfies at least half of the non-tree edges, while both satisfy all tree edges. We choose our initial numbering $\nu_0(\cdot)$ to be the one that satisfies more edges.

This computation of $\nu_0(\cdot)$ is easily carried out I/O-efficiently. After sorting the edges of G by their heads, a scan of this edge list suffices to choose one in-edge for each vertex and, if there is more than one vertex without in-edges, add a new source s and connect it to each such vertex. Thus, T_0 can be constructed using $\mathrm{O}(\mathrm{sort}(m))$ I/Os, where $\mathrm{sort}(N) = \Theta\left(\frac{N}{B}\log_{M/B}\frac{N}{B}\right)$ is the I/O complexity of sorting N elements [1]. The numberings $\nu_l(\cdot)$ and $\nu_r(\cdot)$ are easily computed by computing an Euler tour and applying list ranking to the computed tour [9], which takes $\mathrm{O}(\mathrm{sort}(n))$ I/Os. Then it suffices to sort and scan the vertex and edge sets of G to label every edge with the numbers assigned to its endpoints by $\nu_l(\cdot)$ and $\nu_r(\cdot)$, and count the edges satisfied by each numbering, in order to choose $\nu_0(\cdot)$. In summary, the initialization step of our algorithm takes $\mathrm{O}(\mathrm{sort}(m))$ I/Os.

2.2 Growing the Satisfied Subgraph. Each iteration of the algorithm now computes a new numbering $\nu_i(\cdot)$ from the current numbering $\nu_{i-1}(\cdot)$ so that $\nu_i(\cdot)$ satisfies strictly more edges than $\nu_{i-1}(\cdot)$. We do this in two phases. In the first phase, we compute an out-tree T_i of s and a numbering $\nu'_i(\cdot)$ that satisfies every edge in T_i and such that $\nu'_i(x) \geq \nu_{i-1}(x)$, for all $x \in V$. In the second phase, we compute a numbering $\nu''_i(\cdot)$ by

processing the subgraph $G_{\nu_{i-1}}$ of G satisfied by $\nu_{i-1}(\cdot)$. This numbering satisfies all edges of $G_{\nu_{i-1}}$ and has the property that $\nu_i''(x) \geq \nu_i'(x)$, for all $x \in V$. We obtain the new numbering $\nu_i(\cdot)$ by ordering the vertices in G according to $\nu_i''(\cdot)$ and then numbering the vertices of G in order.[1] Next we describe the computation of $\nu_i'(\cdot)$, $\nu_i''(\cdot)$, and $\nu_i(\cdot)$ in detail.

Computing $\nu_i'(\cdot)$. To construct the tree T_i, we proceed similar to the construction of T_0, choosing one in-edge yx per vertex $x \neq s$ to be included in T_i. This time, however, we choose each such edge yx so that $\nu_{i-1}(y)$ is maximized; see Figure 1(d). Similar to the construction of T_0, this construction can be carried out by sorting the edges of G by their heads and then scanning the edge list to choose the in-edge of each vertex to be included in T_i. (Recall that each edge yx is labelled with the numbers $\nu_{i-1}(y)$ and $\nu_{i-1}(x)$ of its endpoints, making it easy to identify the in-edge yx of each vertex x that maximizes $\nu_{i-1}(y)$.) Next we construct an Euler tour of T_i and apply list ranking to compute a preorder numbering of T_i, which is also a topological ordering of T_i. We sort the vertices of T_i in this order and then apply time-forward processing to compute, for every vertex $x \in T_i$, $\nu_i'(x) := \max(\nu_{i-1}(x), \nu_i'(p_i(x)) + 1)$, where $p_i(x)$ denotes x's parent in T_i; see Figure 1(e). The sorting and scanning of the vertex and edge sets of G, and the application of the Euler tour technique, list ranking, and time-forward processing to T_i take $O(\mathrm{sort}(m))$ I/Os in total.

Computing $\nu_i''(\cdot)$. In the second step, we sort the vertices according to $\nu_{i-1}(\cdot)$ and the edges of $G_{\nu_{i-1}}$ by their tails. Then we apply time-forward processing to $G_{\nu_{i-1}}$, which is possible because $\nu_{i-1}(\cdot)$ defines a topological ordering of $G_{\nu_{i-1}}$ (by definition, $\nu_{i-1}(\cdot)$ satisfies all edges in $G_{\nu_{i-1}}$). For every vertex, we compute $\nu_i''(x) := \max(\{\nu_i'(x)\} \cup \{\nu_i''(y) + 1 \mid yx \in G_{\nu_{i-1}}\})$, which ensures that $\nu_i''(\cdot)$ satisfies every edge of $G_{\nu_{i-1}}$; see Figure 1(f). This takes $O(\mathrm{sort}(m))$ I/Os.

Computing $\nu_i(\cdot)$. To prepare for the next iteration, we compute $\nu_i(\cdot)$ by sorting the vertices in G by $\nu_i''(\cdot)$ and then numbering them in order; see Figure 1(g). Using a constant number of sorting and scanning passes, we label every edge with the numbers of its endpoints and accordingly classify the edge as satisfied or violated. This takes $O(\mathrm{sort}(m))$ I/Os.

2.3 Analysis. From the above discussion, it follows that the initialization and each iteration of the algorithm take $O(\mathrm{sort}(m))$ I/Os. Thus, the I/O complexity of the whole algorithm depends on the number of iterations the algorithm executes. The following lemma bounds this number of iterations.

LEMMA 2.1. ITERTS *takes at most $l-1$ iterations to satisfy all edges in G, where l is the length of the longest path in G.*

Proof. For a vertex x, let l-dist(x) be the length of the longest path from s to x in G. We prove by induction on i that $\nu_{i-1}(\cdot)$ satisfies all in-edges of vertices x with l-dist$(x) \leq i+1$. Thus, if l denotes the length of the longest path in G, $\nu_{l-1}(\cdot)$ satisfies all edges of G.

The base case, $i = 0$, is trivial because $\nu_0(s) = 1$, while $\nu_0(x) > 1$, for all $x \neq s$. Hence, all out-edges of s are satisfied by $\nu_0(\cdot)$, which is a superset of the in-edges of all vertices x with l-dist$(x) \leq 1$.

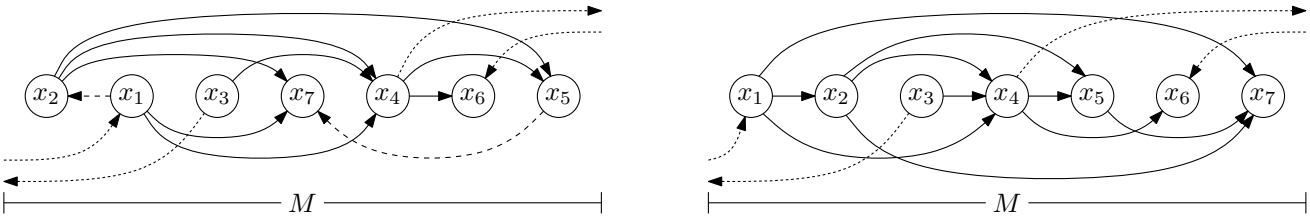
So assume the claim holds for $i < k$. We need to prove it for $i = k$. It suffices to prove that $\nu_k''(\cdot)$ satisfies all in-edges of vertices x with l-dist$(x) \leq k+1$ because $\nu_k(\cdot)$ is obtained by ordering the vertices by $\nu_k''(\cdot)$ and then numbering them in order. In particular, $\nu_k''(x) < \nu_k''(y)$ implies $\nu_k(x) < \nu_k(y)$, and $\nu_k(\cdot)$ satisfies all edges satisfied by $\nu_k''(\cdot)$.

First we prove that $\nu_k''(x) = \nu_k'(x) = \nu_{k-1}(x)$, for all x with l-dist$(x) \leq k$. Since every in-neighbour y of such a vertex x satisfies l-dist$(y) \leq k$ and $\nu_{k-1}(x)$ satisfies every in-edge of x, this implies that $\nu_k''(y) = \nu_{k-1}(y) < \nu_{k-1}(x) = \nu_k''(x)$, that is, $\nu_k''(\cdot)$ satisfies the in-edges of all vertices x with l-dist$(x) \leq k$. We prove our claim by induction on l-dist(x).

For l-dist$(x) = 0$, we have $x = s$ and $\nu_k''(s) = \nu_k'(s) = \nu_{k-1}(s) = 1$ because s is the source of $G_{\nu_{k-1}}$ and the root of T_{k-1}. For $0 <$ l-dist$(x) \leq k$, we have $\nu_k'(x) = \max(\nu_{k-1}(x), \nu_k'(p_k(x)) + 1)$. However, we have l-dist$(p_k(x)) <$ l-dist(x) and, hence, $\nu_k'(p_k(x)) = \nu_{k-1}(p_k(x))$. Furthermore, $\nu_{k-1}(p_k(x)) < \nu_{k-1}(x)$ because $p_k(x)$ is an in-neighbour of x and $\nu_{k-1}(\cdot)$ satisfies all in-edges of x. This implies that $\nu_k'(x) = \nu_{k-1}(x)$. Similarly, we have $\nu_k''(x) = \max(\{\nu_k'(x)\} \cup \{\nu_k''(y) + 1 \mid yx \in G_{\nu_{k-1}}\})$. Every in-neighbour y of x in $G_{\nu_{k-1}}$ satisfies l-dist$(y) <$ l-dist(x). Hence, by the induction hypothesis and because $\nu_{k-1}(\cdot)$ satisfies the edge yx, $\nu_k''(y) = \nu_k'(y) = \nu_{k-1}(y) < \nu_{k-1}(x) = \nu_k'(x)$, and $\nu_k''(x) = \nu_k'(x) = \nu_{k-1}(x)$.

To complete the proof, we need to show that $\nu_k''(\cdot)$ satisfies all in-edges of vertices x with l-dist$(x) = k+1$. Consider such a vertex x, and let y be an in-neighbour of x. The parent $p_k(x)$ of x in T_k is chosen so that $\nu_{k-1}(p_k(x)) \geq \nu_{k-1}(y)$. Hence, $\nu_k'(x) \geq \nu_k'(p_k(x)) + 1 = \nu_{k-1}(p_k(x)) + 1 \geq \nu_{k-1}(y) + 1$. We also have $\nu_k''(x) \geq \nu_k'(x)$, that is, $\nu_k''(x) > \nu_{k-1}(y)$. On the other hand, since y is an in-neighbour of x, we have l-dist$(y) \leq k$ and, hence, $\nu_k''(y) = \nu_{k-1}(y)$. Thus, the edge yx is satisfied by $\nu_k''(\cdot)$. Since this argument applies to all in-

[1] The orderings defined by $\nu_i(\cdot)$ and $\nu_i''(\cdot)$ are identical, but $\nu_i''(\cdot)$ may not assign unique numbers to vertices and may assign numbers greater than N.



(a) Subgraph before processing.

(b) Subgraph after processing.

Figure 2: Local topological ordering of memory-sized subgraphs. In (a), vertices are arranged by $\nu_i''(\cdot)$. Solid edges are satisfied edges in the subgraph, dashed edges are violated edges in the subgraph, and dotted edges have only one endpoint in the subgraph, that is, are not local. In (b), the vertices are rearranged to ensure that all edges local to the subgraph are satisfied.

edges of vertices x with l-dist$(x) = k + 1$, and we have already shown that $\nu_k''(\cdot)$ satisfies all in-edges of vertices x with l-dist$(x) \leq k$, this finishes the proof. □

By Lemma 2.1, IterTS is guaranteed to terminate, after at most $n - 2$ iterations. For many graphs, the longest path has length significantly less than $n - 1$, guaranteeing a faster termination of the algorithm. Even for graphs with long paths, our experiments show that, in practise, IterTS terminates much faster than predicted by Lemma 2.1.

2.4 Satisfying Local Edges. By our analysis in the previous subsection, the cost of our algorithm depends crucially on the number of iterations it needs to satisfy all edges in the DAG. In this section, we discuss a heuristic that helped us reduce the number of iterations significantly.

The idea is to immediately satisfy violated edges whose endpoints are "not too far apart" in the current ordering. To this end, we add the following step between sorting the vertices of G by $\nu_i''(\cdot)$ and numbering them in order to compute $\nu_i(\cdot)$. Let V_i be the list of vertices of G, sorted by $\nu_i''(\cdot)$. We greedily break V_i into contiguous sublists $V_{i,1}, V_{i,2}, \ldots, V_{i,q}$ so that the subgraph $G[V_{i,j}]$ induced by each sublist $V_{i,j}$ fits in memory. We load each such subgraph $G[V_{i,j}]$ into memory and compute a topological ordering of its vertices, thereby producing a new ordered list $V'_{i,j}$ of the vertices in $G[V_{i,j}]$. We concatenate these lists $V'_{i,1}, V'_{i,2}, \ldots, V'_{i,q}$ to obtain a new ordered vertex list V'_i of G and then compute $\nu_i(\cdot)$ by numbering the vertices in V'_i in order. Since two vertices in different subgraphs $G[V_{i,j}]$ and $G[V_{i,k}]$ appear in the same relative order in V_i and V'_i, this strategy ensures that $\nu_i(\cdot)$ satisfies all edges within each memory-sized subgraph $G[V_{i,j}]$, while also satisfying edges between subgraphs $G[V_{i,j}]$ and $G[V_{i,k}]$ that were satisfied by $\nu_i''(\cdot)$. In other words, using this heuristic, the edges satisfied by $\nu_i(\cdot)$ are a superset of the edges satisfied by $\nu_i''(\cdot)$. This is illustrated in Figure 2.

To implement this strategy, we label every edge of G with the numbers assigned to its endpoints by $\nu_i''(\cdot)$. Then we produce the list V_i by sorting the vertices of G by $\nu_i''(\cdot)$, using their vertex IDs as tie breakers. Since this defines a total order on the vertices of G, it suffices to inspect the labels of the endpoints of each edge to determine which endpoint occurs later in V_i. We call this the *high* endpoint, and the other endpoint the *low* endpoint of the edge. We sort the edges of G by their high endpoints. Then we scan V_i and the sorted edge list to partition V_i into sublists $V_{i,1}, V_{i,2}, \ldots, V_{i,q}$ and, simultaneously, construct the edge lists $E_{i,1}, E_{i,2}, \ldots, E_{i,q}$ of the graphs $G[V_{i,1}], G[V_{i,2}], \ldots, G[V_{i,q}]$. During this scan, when considering a vertex x for inclusion in the current sublist $V_{i,j}$, we inspect all edges with high endpoint x. Such an edge yx has both its endpoints in $G[V_{i,j} \cup \{x\}]$ if and only if its low endpoint y succeeds the first vertex z of $V_{i,j}$ in V_i, which can be determined using a simple comparison of the labels of y and z. Thus, we can count these edges by scanning the edges with x as their high endpoints and add the count to the size of $G[V_{i,j}]$ to determine the size of $G[V_{i,j} \cup \{x\}]$. If $G[V_{i,j} \cup \{x\}]$ has size at most M, we add x to $V_{i,j}$ and all edges with high endpoint x and low endpoint in $V_{i,j}$ to $E_{i,j}$. Otherwise x becomes the first vertex in the next sublist $V_{i,j+1}$, and $E_{i,j+1}$ is initially empty. Then we proceed to the next vertex in V_i.

Once we have constructed the vertex and edge lists $V_{i,1}, V_{i,2}, \ldots, V_{i,q}$ and $E_{i,1}, E_{i,2}, \ldots, E_{i,q}$ in this manner, we load the graphs $G[V_{i,1}], G[V_{i,2}], \ldots, G[V_{i,q}]$ into memory, one graph at a time. For each such graph $G[V_{i,j}]$, we compute its topologically sorted vertex list $V'_{i,j}$ in memory and append it to the vertex list V'_i.

Since the cost of this procedure is dominated by the cost of producing the initial sorted vertex and edge lists, this heuristic adds O(sort(m)) I/Os to the cost of each iteration.

3 Other Approaches to Topological Sorting

There are other approaches to topological sorting that are worth considering, as they are either natural or were proposed with I/O efficiency or parallelism in mind and, thus, may achieve better performance than IterTS, at least on certain inputs. In our experiments, we compared the performance of these algorithms to the performance of IterTS.

3.1 Topological Sorting Using Semi-External DFS (SeTS). A classical method for topological sorting is to perform DFS on the DAG and number the vertices in reverse postorder [10]. Using this strategy on top of the semi-external DFS heuristic of [18], one obtains an algorithm for topological sorting that should be very efficient as long as the vertex set of the graph fits in memory.

3.2 Iterative Peeling of Sources and Sinks (PeelTS). Another classical method for topological sorting is to iteratively remove sources and sinks. The algorithm starts with the graph $G_0 := G$. The ith iteration identifies all sources and sinks of the current graph G_{i-1} and numbers them, sources up from 1, sinks down from N. Then these vertices are removed, which produces a new subgraph G_i whose sources and sinks are numbered in the next iteration. The algorithm terminates as soon as the current graph G_{i-1} is empty.

A naive implementation of this strategy requires one random access per edge to test, for each neighbour of a removed vertex, whether it becomes a source or sink as a result of this removal and, thus, should be numbered and removed in the next iteration. In our experiments, we used the following, more I/O-efficient implementation.

As in IterTS, we assume the initial DAG has only one source. We start by arranging the vertices of G and their adjacency lists in an order that attempts to approximate the order they are numbered by PeelTS, in order to be able to identify sources and sinks in each peeling round by scanning this sorted list instead of using random accesses. To this end, we compute an out-tree T of the source as in Section 2, and we label the vertices of G with their in- and out-degrees in G and with their depths in T. This information can be computed using the Euler tour technique and list ranking. Now we sort the vertices and their adjacency lists by their depths in T. Let L be the resulting list.

Having preprocessed G in this manner, we start the process of iteratively removing sources and sinks. The ith iteration of this process requires four lists V_{i-1}^-, V_{i-1}^+, E_{i-1}^-, and E_{i-1}^+ as inputs. These lists respectively contain the sources of G_{i-1}, the sinks of G_{i-1}, the out-edges in G_{i-1} of all vertices in V_{i-1}^-, and the in-edges in G_{i-1} of all vertices in V_{i-1}^+. The lists V_0^-, V_0^+, E_0^-, and E_0^+ required by the first iteration are easily computed by scanning L.

Now consider the computation in the ith iteration. Numbering the sources and sinks in V_{i-1}^- and V_{i-1}^+ is a simple matter of scanning these two lists. To construct V_i^- and V_i^+, we sort the edges in E_{i-1}^- by their heads, and the edges in E_{i-1}^+ by their tails. Now we scan E_{i-1}^- forward. For every edge $xy \in E_{i-1}^-$, we decrease the in-degree of vertex y in L by one. If y's in-degree is now 0, we mark y and its adjacency list as deleted in L, append y to V_i^-, and append y's out-edges to E_i^-. By the ordering of the edges in E_{i-1}^-, locating the heads of the edges in E_{i-1}^- in L is a matter of scanning L forward once until we have found all heads of edges in E_{i-1}^-. After processing the edges in E_{i-1}^- in this manner, we process the edges in E_{i-1}^+ similarly, with the exception that we scan E_{i-1}^+ and L backward, and we decrease the out-degrees of the tails of the edges in E_{i-1}^+.

After a number of iterations, deleted elements start to accumulate in L, contributing unnecessarily to the cost of scanning L. To reduce the scanning cost of L, we compact L periodically. For some load factor $0 < \alpha < 1$, we call a sublist of L *α-sparse* if more than a $(1-\alpha)$-fraction of the elements in the sublist are marked as deleted. In each iteration, we find the longest α-sparse prefix of the prefix of L scanned in this iteration, and the longest α-sparse suffix of the suffix of L scanned in this iteration, and we compact these two sublists by storing the unprocessed elements in them consecutively. In our implementation, we chose $\alpha = 5\%$, which we determined experimentally gave the best performance.

3.3 Divide and Conquer Based on Reachability Queries (ReachTS). In [17], a parallel divide-and-conquer algorithm for topological sorting based on reachability queries is described. We implemented an external-memory version of this algorithm.

If the DAG fits in memory, we load it into memory and sort it. Otherwise, we apply the following partitioning strategy. We arrange the vertices in a random order $x_1, x_2, \ldots, x_n$. Then we use binary search to find the lowest index k such that vertices $x_1, x_2, \ldots, x_k$ can reach at least $n/2$ vertices in the DAG. Let A be the set of vertices reachable from $x_1, x_2, \ldots, x_{k-1}$, and let B be the set of vertices reachable from x_k. The algorithm now recursively sorts the vertices in the sets $V \setminus (A \cup B)$, $A \setminus B$, $\{x_k\}$, $B \setminus A$, and $A \cap B$ and concatenates the result; see Figure 3. The correctness of this strategy is shown in [17]. It is also shown that the expected size of each set is $n/2$, making this algorithm terminate after

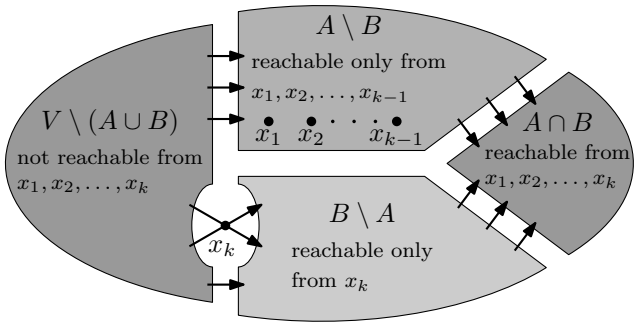


Figure 3: The five parts of G ReachTS recurses on.

expected $\log n$ levels of recursion.

To find the set of vertices reachable from a set S during the binary search that finds the index k, we use an implementation of directed breadth-first search. We start by initializing two sets $L := S$ and $R := S$. The set L is the current BFS level. The set $R \supseteq L$ is the set of vertices already seen by the BFS. Then we proceed in iterations. In each iteration, we compute the next BFS level L' as the set of out-neighbours of the vertices in L that are not in R. Then we set $R := R \cup L'$ and $L := L'$ for the next iteration. We repeat this until $L = \emptyset$. At this point, the set R is the set of vertices reachable from S. Each iteration of this directed BFS procedure can be implemented using $O(sort(m))$ I/Os. The set L' can be computed by scanning L and the set of edges of G to find all out-edges of vertices in L. Then we sort the set of heads of these edges and scan the resulting sorted list and R to remove all duplicates and vertices that belong to R from the list. The result is L'. Since each BFS iteration takes $O(\text{sort}(m))$ I/Os, ReachTS should be efficient if the "diameter" of the graph is low.

4 Implementation and Experiments

In this section, we discuss some choices we made in our implementations of the different algorithms, the environment and data sets we used to evaluate the algorithms, and the results we obtained in our experiments.

4.1 Implementation. We implemented IterTS, PeelTS and ReachTS in C++ and using the STXXL library [12], which is an implementation of the C++ STL for external memory computations. For SeTS, we used an implementation provided by Andreas Beckmann [7].

We used STXXL vectors to store the vertex and edge sets of the graph. All sorting steps in our implementation were accomplished using the STXXL sorting algorithm. The implementation of time-forward processing requires a priority queue, for which we used the one provided in STXXL.

We used the standard construction of an Euler tour of a tree, which generates a list of edges incident to each vertex by duplicating each edge and then sorting the edge list. Then a scan of this sorted list suffices to generate the Euler tour [9]. Thus, an Euler tour is easily constructed using STXXL primitives.

For list ranking, we used an algorithm of [19]. Ajwani et al. [2] provided an STXXL-based implementation of this algorithm as part of their undirected BFS implementation, and we re-used this code.

4.2 Test Environment. All experiments were run on a PC with a 3.33GHz Intel Core i5 processor, 4GB of RAM, and one 500GB 7200RPM IDE disk using the XFS file system. The operating system was Ubuntu 9.10 Linux with a 2.6.31 Linux kernel. The code was compiled using g++ 4.4.1 and optimization level –O3. For our experiments, we limited the available RAM to 1GB (using the `mem=` kernel option). All of our timing results refer to wall clock times in hours.

4.3 Data Sets. We tested the algorithms on synthetic graphs chosen with certain characteristics that should be hard or easy for different algorithms among the ones we implemented. We also ran the algorithms on real web graphs with their edges redirected to ensure the graphs are acyclic. The number of vertices in the graphs were between 2^{25} and 2^{28}, the number of edges between 2^{27} and 2^{30}. The following is the list of graph classes we used to evaluate the algorithms.

Random: We generated these graphs according to the $G_{n,m}$ model; that is, we generated m edges, choosing each edge endpoint uniformly at random from a set of n vertices. The edges were directed from lower to higher endpoints.

Width-one: To construct these graphs, we started with a long path of $n-1$ edges. Then we added $m-n+1$ random edges according to the $G_{n,m}$ model as for random graphs.

Layered: These graphs were constructed from $\sqrt{n}$ layers of $\sqrt{n}$ vertices, with random edges between adjacent layers. To generate these graphs, we first chose, for each vertex in a given layer, a random in-neighbour in the previous layer and a random out-neighbour in the next layer. Then we added more random edges between adjacent layers to increase the edge count to m.

Semi-layered: Layered graphs consist of many moderately long paths but are too structured, which makes them extremely easy inputs for PeelTS.

Semi-layered graphs aim to have moderately long paths but with less structure. To construct these graphs, we first constructed $q := n^{1/3}$ layered DAGs $G_1, G_2, \ldots, G_q$ consisting of $n^{1/3}$ layers of size $n^{1/3}$ each. Then we added random edges between the DAGs by generating random quadruples (i, j, h, k) with $i < j$ and $h > k$ and, for each such quadruple, adding a random edge from layer h of G_i to layer k of G_j.

Low-width: These graphs were constructed in the same way as layered graphs. However, the number of layers was set to $1,000,000$ in this case and the size of a layer was set to $n/1,000,000$. Moreover, in the first phase of the construction of the graph, which chooses one in- and one out-neighbour per vertex, we connected the ith node in the jth layer to the ith node in the $(j+1)$st layer, thereby starting with $n/1,000,000$ disjoint paths of length $1,000,000$. Then we added random edges between layers as for layered graphs.

Grid: These graphs were formed by taking a $\sqrt{n} \times \sqrt{n}$ grid and directing all horizontal edges to the right and all vertical edges down.

Webgraphs: The web graphs were produced by real web crawls of the .uk domain, the .it domain, and from data produced by a more global crawl using the Stanford WebBase crawler. They were obtained from `http://webgraph.dsi.unimi.it/`. Since these graphs were not necessarily acyclic, we redirected the edges from lower vertex IDs to higher vertex IDs.

4.4 Experimental Results. The main goal of our experiments was to compare the algorithms, study how they are affected by the structure of the input graph, and use the results to recommend which algorithm to use if there is a-priory knowledge of the graph structure. Table 1 shows the running times of the algorithms on different input graphs. In order to bound the time spent on our experiments, we used the following rules. (1) Each algorithm was given an amount of time at least 10 times the time used by IterTS to process the same input. If it did not produce a result in the allocated time, we terminated it. This is indicated by dashes in the table, with superscripts indicating the amount of time given to the algorithm. (2) If IterTS took more than one day to process an input and was consistently faster than the other algorithms on smaller inputs, we did not run the other algorithms on this input. This is indicated by stars in the table. (3) Since SeTS is a semi-external algorithm and 2^{26} vertices do not fit in 1GB of memory, we did not run it on larger inputs if it did not finish in the allocated time on the smallest input with 2^{26} vertices (which was the case for all input types).

4.4.1 Comparison of Running Times. With the exception of the second random graph instance, IterTS outperformed PeelTS and ReachTS. As expected, random graphs proved to be easy instances for all algorithms, with usually a factor of less than two between the running times of IterTS, PeelTS, and ReachTS. On most of the other inputs, PeelTS and ReachTS were not able to process any of the inputs in the allotted amount of time, that is, IterTS outperformed them by at least one order of magnitude on these inputs. PeelTS was able to process all layered and grid graph instances we tried. For grid graphs, the running time was still more than 10 times higher than that of IterTS. Layered graphs are a particularly easy input for PeelTS because the preprocessing stage of the algorithm ends up arranging the vertices layer by layer, which is also the order in which the peeling phase peels sources and sinks. Thus, each peeling round scans exactly those vertices that are removed from the graph in this round. We created semi-layered graphs to eliminate this effect and, as expected, the performance of PeelTS broke down on these graphs. ReachTS performed better on semi-layered graphs than on layered graphs. We believe this was a result of somewhat shorter shortest paths in the semi-layered graphs, which made the reachability queries in ReachTS cheaper.

The results on web graphs presented a surprise, with ReachTS being able to process one of these graphs in 4 times the time taken by IterTS, while not being able to process the bigger web graphs. PeelTS was not able to process any of these graphs in the allotted time. This is surprising because we expected these graphs to behave similarly to random graphs, particularly given that the edge directions were essentially chosen randomly. Thus, these graphs should not have posed any challenges.

On inputs whose vertex sets fit in memory, SeTS outperformed IterTS on most inputs, while IterTS was faster on some inputs. Width-one graphs turned out to be particularly easy instances for SeTS. On these inputs, it was nearly two orders of magnitude faster than IterTS. This concurs with the discussion in [18], where it was stated that the semi-external DFS algorithm performs very well for deep DFS trees. As expected, the comparison between IterTS and SeTS changed dramatically once the graph's vertex set did not fit in memory any more. SeTS was not able to process any of these inputs within the allotted time, that is, IterTS outperformed SeTS by at least one order of magnitude.

In summary, we conclude that SeTS is the algorithm that should be used for semi-external inputs, while IterTS is the clear winner on larger inputs. PeelTS and ReachTS were not competitive with either SeTS or IterTS.

4.4.2 The Effect of the Graph's Structure. Recall from Section 2.3 that the running time of IterTS is determined mostly by the number of iterations it needs to satisfy all edges in the graph. With the exception of width-one graphs and the larger semi-layered graphs, the number of iterations needed by IterTS was low, even though the graph structure had some impact on the number of iterations needed. Thus, the performance of IterTS can be considered fairly robust and *almost* independent of the graph's structure. Width-one graphs and the larger semi-layered graphs posed a greater challenge. However, the upper bound on the number of iterations provided by Lemma 2.1 is between 2^{25} and 2^{28} for the input graphs we tested, while IterTS needed less than 20 iterations for all of these inputs and was able to process all our input instances in a reasonable amount of time.

SeTS can be considered equally robust on semi-external instances, even though it benefits from deep DFS trees, as already discussed. In contrast, IterTS benefits from graphs having short paths, even according to the pessimistic prediction of Lemma 2.1. Hence, IterTS is competitive with SeTS, for instance, on semi-external random inputs, while SeTS is significantly faster on width-one graphs.

The other algorithms are much more sensitive to graph structure. By definition, PeelTS needs a large number of peeling rounds for graphs with long paths. For example, for the smallest low-width graph, only 5% of the vertices had been removed after 92,000 peeling rounds, while PeelTS finished after between 73 and 148 rounds for random graphs. On layered graphs, PeelTS also needed a large number (2898–8194) of rounds. The reason for the good performance of PeelTS on these graphs is that the total cost of the rounds is proportional to the total number of vertices, due to the particular order in which the preprocessing phase arranges the vertices. The same should be true for low-width graphs, which are layered graphs with many small layers. The reason why PeelTS was not able to process them was the large number of peeling rounds, each of which incurred some overhead leading to a cost of 1–5s per peeling round. This overhead could have been eliminated for these graphs, given our knowledge of the graph structure, but our goal was not to design customized algorithms for individual graph classes.

ReachTS should perform well on graphs with low diameter and poorly on graphs with long shortest paths, as the most costly part of the algorithm is the BFS-based reachability queries. This intuition is confirmed by its good performance on random graphs and its poor performance on layered, low-width, and grid graphs. For example, the maximum number of BFS levels observed in any reachability query on the random instances was 39, while the smallest low-width graph led to reachability queries with over 1,400 BFS levels before the algorithm was terminated. The performance on semi-layered and width-one graphs, however, contradicts this intuition. Width-one graphs are random graphs, apart from the one path visiting all vertices. So most shortest paths should be short, and the algorithm should perform well, but it did not manage to process any of these instances. Conversely, semi-layered graphs should have fairly long shortest paths; yet, the algorithm performed fairly well on these graphs.

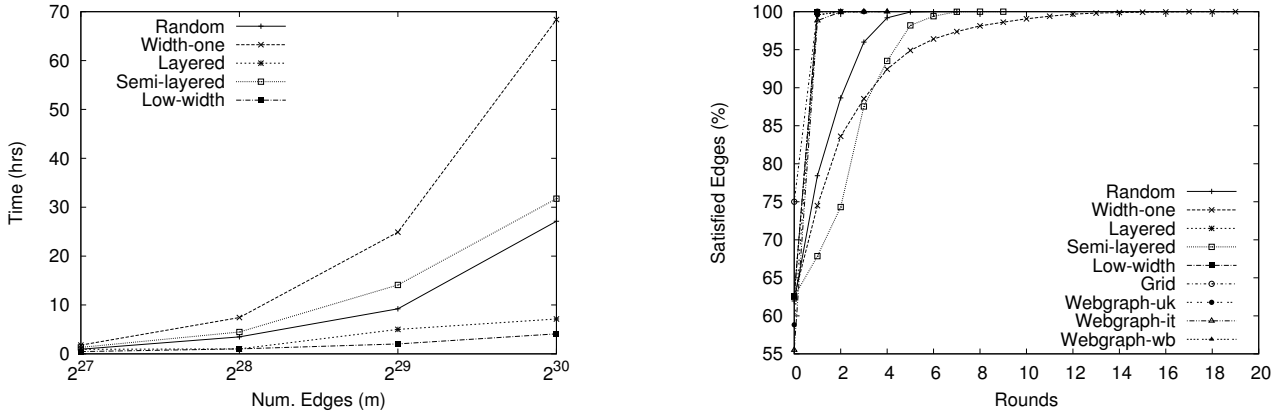
4.4.3 Further Analysis of IterTS. Figure 4(a) shows the running time of IterTS on graphs of different types and sizes but with fixed density. As expected, the running time increased linearly with the input size for layered and low-width graphs, as the number of iterations is nearly independent of the size of the graph. For random, width-one, and semi-layered graphs, the number of iterations required by the algorithm to terminate increased with the input size, leading to a super-linear dependence of the algorithm on the input size.

Another interesting factor to consider is how quickly the satisfied subgraph G_ν converged to the whole DAG G. Figure 4(b) shows the percentage of satisfied edges as a function of the iteration number for the largest input of each type. As can be seen, with the exception of width-one graphs, the algorithm took only few iterations to satisfy nearly all edges. Even for width-one graphs, 95% of the edges were satisfied after only 6 iterations, and nearly 100% were satisfied after 10 iterations. This implies that, under reasonable assumptions about the ratio between the sizes of main memory and disk, the edges that remained violated after 8–10 iterations fit in memory. It would be helpful to switch to an alternate strategy at this point, which takes advantage of this fact in order to avoid a large number of iterations to satisfy the remaining edges. We did not succeed in finding such a strategy.

Our final comment concerns the effect of the local reordering heuristic described in Section 2.4 on the running time of the algorithm. It became clear relatively early on that this heuristic speeds up the algorithm tremendously. So we did not run IterTS without the heuristic, except on some of the smaller inputs. For

Graph class	n	m	m/n	IterTS		PeelTS	ReachTS	SeTS
				Iterations	Time (h)	Time (h)	Time (h)	Time (h)
Random	2^{25}	2^{27}	4	2	0.94	2.71	2.20	0.50
Random	2^{25}	2^{28}	8	5	3.50	8.58	2.39	1.56
Random	2^{26}	2^{28}	4	4	3.47	5.48	4.23	—[2]
Random	2^{26}	2^{29}	8	5	7.48	17.76	10.78	***
Random	2^{27}	2^{29}	4	5	9.22	14.02	9.80	***
Random	2^{28}	2^{30}	4	7	27.13	***	***	***
Width-one	2^{25}	2^{27}	4	4	1.78	—[1]	—[0]	0.05
Width-one	2^{25}	2^{28}	8	6	4.25	—[2]	—[2]	0.08
Width-one	2^{26}	2^{28}	4	8	7.42	—[3]	—[3]	—[3]
Width-one	2^{26}	2^{29}	8	9	13.46	—[6]	—[6]	***
Width-one	2^{27}	2^{29}	4	14	24.90	***	***	***
Width-one	2^{28}	2^{30}	4	19	68.38	***	***	***
Layered	2^{25}	2^{27}	4	2	0.92	2.70	—[0]	0.48
Layered	2^{25}	2^{28}	8	1	0.76	4.62	—[1]	1.17
Layered	2^{26}	2^{28}	4	1	1.02	6.33	—[1]	—[1]
Layered	2^{26}	2^{29}	8	1	1.49	10.76	—[1]	***
Layered	2^{27}	2^{29}	4	3	5.01	25.55	—[2]	***
Layered	2^{28}	2^{30}	4	2	7.14	57.87	—[3]	***
Semi-layered	2^{25}	2^{27}	4	3	1.33	—[1]	2.58	0.34
Semi-layered	2^{25}	2^{28}	8	5	3.26	—[2]	8.02	0.75
Semi-layered	2^{26}	2^{28}	4	5	4.47	—[2]	15.77	—[2]
Semi-layered	2^{26}	2^{29}	8	7	10.08	—[4]	20.83	***
Semi-layered	2^{27}	2^{29}	4	8	14.09	—[5]	66.97	***
Semi-layered	2^{28}	2^{30}	4	9	31.75	***	***	***
Low-width	2^{25}	2^{27}	4	1	0.47	—[1]	—[1]	0.35
Low-width	2^{25}	2^{28}	8	1	0.72	—[1]	—[1]	0.93
Low-width	2^{26}	2^{28}	4	1	1.01	—[1]	—[1]	—[1]
Low-width	2^{26}	2^{29}	8	1	1.48	—[1]	—[1]	***
Low-width	2^{27}	2^{29}	4	1	2.03	—[1]	—[1]	***
Low-width	2^{28}	2^{30}	4	1	4.09	—[2]	—[2]	***
Grid	2^{25}	$\approx 2^{26}$	2	1	0.31	4.14	—[1]	0.50
Grid	2^{26}	$\approx 2^{27}$	2	1	0.67	8.46	—[1]	—[1]
Grid	2^{27}	$\approx 2^{28}$	2	1	1.38	18.67	—[1]	***
Grid	2^{28}	$\approx 2^{29}$	2	1	2.84	44.54	—[2]	***
Webgraph	18.5m	298.1m	16.1	3	1.88	—[1]	7.75	1.30
Webgraph	41.3m	1,150.7m	25.9	4	9.06	—[4]	—[4]	3.49
Webgraph	118.1m	1,019.9m	8.6	4	10.13	—[4]	—[4]	—[4]

Table 1: Experimental results. Dashes indicate inputs that could not be processed by the algorithm in the allocated time. Superscripts indicate the number of days after which each run was terminated. A superscript of 0 means the run was terminated after 15 hours. Stars indicate that these experiments were not run, following our rules stated at the beginning of Section 4.4.



(a) (b)

Figure 4: (a) Increase of the running times of IterTS for graphs with fixed density $m/n = 4$ and increasing m. (b) Increase of the number of satisfied edges per iteration for graphs with $n = 2^{28}$ and $m = 2^{30}$.

graphs with 2^{25} vertices and 2^{27} edges, we observed a reduction in the number of iterations from between 4 and 21 to between 1 and 3 as a result of the heuristic. The only exceptions were grid graphs, which took one iteration with or without the heuristic, and width-one graphs, which took 4 iterations with the heuristic and which we terminated after 51 iterations without the heuristic.

5 Conclusions

Our experiments demonstrated that IterTS and SeTS substantially outperform PeelTS and ReachTS and are less susceptible to variations in the graph's structure. While SeTS outperformed IterTS on most inputs whose vertex sets fit in memory, IterTS was able to process larger inputs efficiently and SeTS was not. As such, we conclude that IterTS is the first algorithm for topologically sorting large DAGs that can efficiently process graphs whose vertex sets are beyond the main memory size, while SeTS remains the best choice for topologically sorting inputs whose vertex sets fit in memory.

References

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[2] D. Ajwani, U. Meyer, and V. Osipov. Improved external memory BFS implementation. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments*, 2007.

[3] L. Arge, J. S. Chase, P. N. Halpin, L. Toma, J. S. Vitter, D. Urban, and R. Wickremesinghe. Efficient flow computation on massive grid terrain data sets. *Geoinformatica*, 7(4):283–313, 2003.

[4] L. Arge and M. Revsbæk. I/O-efficient contour tree simplification. In *Proceedings of the 20th International Symposium on Algorithms and Computation*, volume 5878 of *Lecture Notes in Computer Science*, pages 1155–1165. Springer-Verlag, 2009.

[5] L. Arge and L. Toma. Simplified external memory algorithms for planar DAGs. In *Proceedings of the 9th Scandinavian Workshop on Algorithm Theory*, volume 3111 of *Lecture Notes on Computer Science*, pages 493–503. Springer-Verlag, 2004.

[6] L. Arge, L. Toma, and N. Zeh. I/O-efficient algorithms for planar digraphs. In *Proceedings of the 15th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 85–93, 2003.

[7] A. Beckmann. Parallelizing semi-external depth first search. Master's thesis, Martin-Luther-Universität, Halle, Germany, 2005.

[8] A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook. On external memory graph traversal. In *Proceedings of the 11th ACM-SIAM Symposium on Discrete Algorithms*, pages 859–860, 2000.

[9] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proceedings of the 6th ACM-SIAM Symposium on Discrete Algorithms*, pages 139–149, 1995.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.

[11] A. Cosgaya-Lozano and N. Zeh. A heuristic strong connectivity algorithm for large graphs. In *Proceedings of the 8th International Symposium on Experimental Algorithms*, volume 5526 of *Lecture Notes in Computer Science*, pages 113–124. Springer-Verlag, 2009.

[12] R. Dementiev, L. Kettner, and P. Sanders. STXXL: Standard library for XXL data sets. *Software: Practice and Experience*, 38(6):589–637, 2007.

[13] A. Maheshwari, M. H. M. Smid, and N. Zeh. I/O-efficient algorithms for computing planar geometric spanners. *Computational Geometry*, 40(3):252–271, 2008.

[14] A. Maheshwari and N. Zeh. I/O-optimal algorithms for outerplanar graphs. *Journal of Graph Algorithms and Applications*, 8(1):47–87, 2004.

[15] A. Maheshwari and N. Zeh. I/O-efficient planar separators. *SIAM Journal on Computing*, 38(3):767–801, 2008.

[16] A. Maheshwari and N. Zeh. I/O-efficient algorithms for graphs of bounded treewidth. *Algorithmica*, 54(3):413–469, 2009.

[17] W. Schudy. Finding strongly connected components in parallel using $O(\log^2 n)$ reachability queries. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 146–151, 2008.

[18] J. Sibeyn, J. Abello, and U. Meyer. Heuristics for semi-external depth-first search on directed graphs. In *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures*, pages 282–292, 2002.

[19] J. F. Sibeyn. From parallel to external list ranking. Technical report MPI-I-97-1-021, Max Planck Institut für Informatik, Saarbrücken, Germany, 1997.

[20] Y. Zhang and M. S. Waterman. An Eulerian path approach to global multiple alignment for DNA sequences. *Journal of Computational Biology*, 10:803–820, 2003.

[21] Y. Zhang and M. S. Waterman. An Eulerian path approach to local multiple alignment for DNA sequences. *Proceedings of the National Academy of Sciences of the United States of America*, 102(5):1285–1290, 2005.